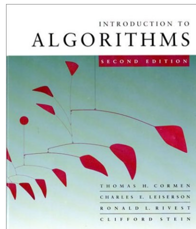
Datastructures and Algorithms



Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein
Introduction to Algorithms (3rd edition)
MIT Press

Red thread (Dutch expression) of this course

Programming requires algorithmic design.

Computers are founded on a universal computation framework.

We will study basic data structures and algorithms.

Some (at first sight simple) problems are undecidable.

Some finite, NP-complete problems, like the *traveling salesman problem*, can probably not be solved efficiently on a computer (if $\mathrm{P} \neq \mathrm{NP}$).

What this course is (not) about

This isn't a programming course (there are no computer labs).

It isn't a theory course either (proofs are omitted or sketched).

Still, this course is crucial to become a good programmer (algorithms, data structures).

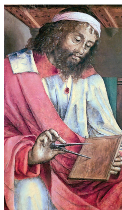
We will delve into the foundations of computer science (undecidability, $\mathrm{P} = \mathrm{NP}$).

And we will look at another model of computation (quantum computing).

Example algorithm: baking a cake

ingredients	input
recipe	algorithm *(software, computer program)*
tools	data structures
oven	hardware
cake	output

Example: Euclid's greatest common divisor algorithm



Euclid (Alexandria, around 300 BC)

Compute the greatest common divisor of two non-negative numbers $n \geq m$:

- if $m = 0$, then return n;
- else, compute the greatest common divisor of m and $n \bmod m$ (i.e., the remainder after dividing n by m).

The second line contains a *recursive call*.

Pseudocode of Euclid's algorithm

Compute greatest common divisor of natural numbers m and n

Euclid(m : natural, n : natural)

if $m = 0$ **then**
  return n
if $n = 0$ **then**
  return m
if $m \leq n$ **then**
  Euclid(m, $n \bmod m$)
else
  Euclid($m \bmod n$, n)

Algorithm

An algorithm is a list of instructions that captures the essence of (part of) a computer program.

Important aspects:

- Termination: does the algorithm always produce output ?
- Correctness: does it meet its requirements ?
- Efficiency: how much time and memory space does it use ?

Our model

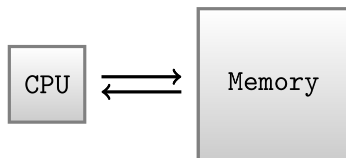
Computer: Random Access Machine (RAM)

Data structure: specification as an abstract data type

Algorithm: description in pseudocode

Random Access Machine (RAM)

Central Processing Unit (CPU) with memory:



Assumptions:

- Unlimited number of memory cells, also called *registers*.
- Primitive operations, like reading or writing to a register, take constant (little) time.

Data structure

A data structure is a particular way of storing and organizing data in a computer so that it can be used efficiently.

Different data structures are suited to different applications; some are highly specialized to specific tasks.

An abstract data type specifies a data structure by operations performed on it, including *exception handling*:

anomalous/exceptional events requiring special processing.

Example: A stack is defined by two operations:

- *push*(d) inserts datum d on top of the stack.
- *pop*() extracts the top item from the stack, *if it is non-empty*.

Pseudocode

Pseudocode resembles a programming language, but is more abstract, and independent of a specific syntax.

- Input and output
- Variable declarations, parameters
- Abstract data types
- Reads from and writes to registers
- Control flow: **if** ... **then** ... (**else** ...) **while** ... **do** ...
  for ... **to** ... **do** ...

  In a computer program this is usually closed, e.g. by **fi** or **od**, but we use indenting instead, which is fine for humans (and Python)
- Recursive calls

Pseudocode constructs

var declares the *types* of the variables used in the pseudocode.

new declares a *fresh* variable inside the pseudocode.

$x \leftarrow v$ *assigns* the value v to variable x.

while b **do** p performs procedure p as long as boolean b is *true* (which is checked every time p terminates).

if b **then** p **else** q performs p if b is *true*, and q if b is *false*.

for $i = 1$ **to** n **do** $p(i)$ sequentially performs $p(1), p(2), \ldots, p(n)$.

for $i = n$ **downto** 1 **do** $p(i)$ performs $p(n), p(n-1), \ldots, p(1)$.

return (x) terminates the program (and returns the value of x).

throw *SomeException* throws some (unspecified) exception if the values of some variables obstruct the computation.

Array

An array $A[1..n]$ consists of n registers $A[1],\ A[2], \ldots, A[n]$.

In the next procedure, n is a parameter of type natural that must be instantiated with a concrete natural number when it is called.

Pseudocode example

Compute maximum among n natural numbers

$FindMax(A[1..n]$: array of naturals)

var $currentMax \leftarrow A[1]$: natural

for $i = 2$ **to** n **do**

if $currentMax < A[i]$ **then**

$currentMax \leftarrow A[i]$

return $currentMax$

Stacks

LIFO: last in first out.

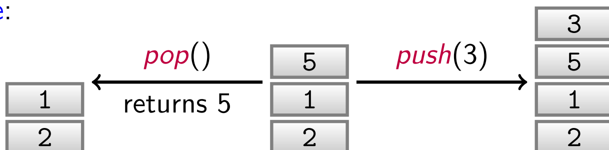
Examples: A stack of plates.

Memory of the undo button in an editor.

Operations: push an element onto the stack.

pop the top element from the stack.

Example:



Stack as an abstract data type

Main operations:

- *push*(d)
  pushes element d onto the top of the stack.
- *pop*()
  removes the top element from the stack and returns it.

Some auxiliary operations:

- *isEmpty*()
  returns a Boolean indicating whether the stack is empty.
- *size*()
  returns the number of elements on the stack.
- *top*()
  returns the top element of the stack *without removing it*.

Exceptions / errors:

- When *pop*() or *top*() is performed on an *empty* stack.

Implementing a bounded stack with an array

The maximum size n of the stack is fixed beforehand,
to determine the number of slots in the array.

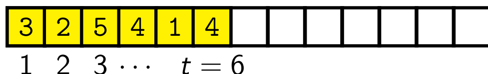
Stack elements are added from left to right in the array.

The value of variable t points at the slot in the array
that holds the top element of the stack.

When the stack is empty, $t = 0$.

When the array overflows, an exception is thrown.

Example: $n = 13$



Bounded stack: pseudocode

Stack implementation with an array $A[1..n]$

var $t \leftarrow 0$: natural

push(d : datum)

```
if t = n then
    throw FullStackException
else
    t ← t + 1
    A[t] ← d
```

pop()

```
if t = 0 then
    throw EmptyStackException
else
    t ← t − 1
    return A[t + 1]
```

Factors for time consumption

Hardware: processor, clock rate, main memory, caches, ...

Software: operating system, programming language, compiler, ...

Algorithm: time complexity

Input size: larger input size tends to mean longer running time

Experimental performance analysis

We can perform experiments, using a real-time clock.

- An implementation is needed.
- For comparisons, we must use the same hardware and software

  (And even then different workloads may induce differences.)
- Only for a finite number of inputs

  (And we may miss inputs with a very bad performance.)
- We don't obtain insight into exhibited performance.

Theoretical time complexity analysis

Worst-case time complexity analysis provides an upper bound, as a function of *the size of the input*.

- No implementation is needed.
- Independent of hardware and software.
- For all possible inputs.
- Provides insight into performance.

But an algorithm with a poor worst-case time complexity may in practice perform fine.

Counting primitive operations for *FindMax*

initialization:

read $A[1]$	1
assign value of $A[1]$ to *currentMax*	1

bookkeeping for loop:

assign value 2 to i	1
check $i \leq n$ ($n - 1$ times true, 1 time false)	n
compute $i + 1$	$n - 1$
assign value of $i + 1$ to i	$n - 1$

body of loop for $i = 2, \ldots, n$:

read *currentMax* and $A[i]$	$2 \cdot (n - 1)$
check whether $currentMax < A[i]$	$n - 1$
possibly assign value of $A[i]$ to *currentMax*	worst-case $n - 1$

return:

return value of *currentMax*	1

Counting primitive operations

Count the number of operations in the worst or average case.

The last case requires a probability distribution over possible executions.

For *FindMax*:

Worst-case: $4 + n + 6 \cdot (n-1) = 7 \cdot n - 2$ operations

Best-case: $4 + n + 5 \cdot (n-1) = 6 \cdot n - 1$ operations

Question: Classify the arrays on which *FindMax* achieves its best and worst performances.

Relating time complexity to input size

Precisely counting the number of operations is tedious.

We are actually most interested in:

- Running time as a function of the size of the input.
- Growth of running time when input size increases.

  (For example, the running time of *FindMax* grows linearly in n.)
- Asymptotic approximation (so that e.g. a constant factor difference between hardware platforms becomes irrelevant).

Big O for upper bounds on complexity

Let $f, g : \mathbb{N} \to \mathbb{R}_{\geq 0}$.

$f \in O(g)$ (i.e., f is in the set $O(g)$) if, for some $C > 0$

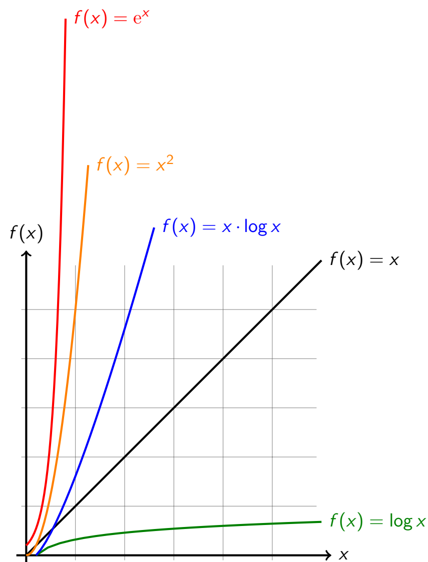
$$f(n) \leq C \cdot g(n) \text{ for all } n \in \mathbb{N}.$$

Example: The (worst- and average-case) time complexity of $FindMax(A[1..n])$ is in $O(n)$.

Examples:

$n^a \in O(n^b)$	for all $0 < a \leq b$
$n^a \in O(b^n)$	for all $a > 0$ and $b > 1$
$\log_a n \in O(n^b)$	for all $a, b > 0$

Growth of some important functions



Divide-and-conquer algorithm

A divide-and-conquer algorithm

- splits the original problem P into smaller subproblems,
- solves the subproblems recursively, and
- combines solutions of subproblems into a solution of P.

Divide-and-conquer algorithms typically have a logarithm
in their time complexity.

Because dividing a problem of input size 2^k into subproblems of size 1 takes k steps.

And $k = \log_2 2^k$.

Tight bounds

$f \in \Theta(g)$ if, and only if, $f \in O(g)$ and $g \in O(f)$.

Example: The time complexity of $\mathit{FindMax}(A[1..n])$ is in $\Theta(n)$.

Question: Relate n^3, n^2, $n^3 + n^2 + 9$, $\log_2 n$, 2^n, $2^{\log_2 n}$, and $\sqrt{n}$.

Queues

FIFO: first in first out.

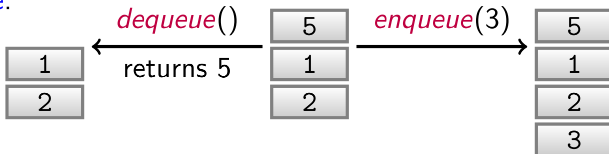
Examples: A bakery.

A printer queue.

Operations: enqueue an element at the *tail* of the queue.

dequeue the element at the *head* of the queue.

Example:



Queue as an abstract data type

Main operations:

- *enqueue*(d)
  adds element d at the tail of the queue.
- *dequeue*()
  removes the element at the head of the queue and returns it.

Some auxiliary operations:

- *isEmpty*()
  returns a Boolean indicating whether the queue is empty.
- *size*()
  returns the number of elements in the queue.
- *head*()
  returns the head element of the queue *without removing it*.

Exceptions / errors:

- When *dequeue*() or *head*() is performed on an *empty* queue.

Queues: example

Operation	Output	(head $\leftarrow Q \leftarrow$ tail)
		()
enqueue(5)	–	(5)
enqueue(3)	–	(5, 3)
head()	5	(5, 3)
size()	2	(5, 3)
dequeue()	5	(3)
isEmpty()	*false*	(3)
dequeue()	3	()
isEmpty()	*true*	()
dequeue()	*EmptyQueueException*	()

Implementing a bounded queue with an array

The maximum size n of the queue is fixed beforehand,
to determine the number of slots in the array.

Queue elements are added from left to right in the array.

The value of variable h points at the slot in the array
that holds the head element of the queue.

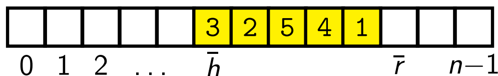
The value of variable r points at the slot in the array
to the right of the tail element of the queue.

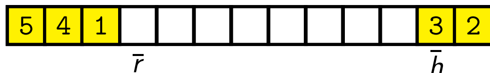
The values of h and r as array slots are interpreted
modulo n (with range $0, \ldots, n-1$).

When the queue is empty ($r = h$) or full ($r = h + n$),
an exception is thrown for a *dequeue* or *enqueue*, respectively.

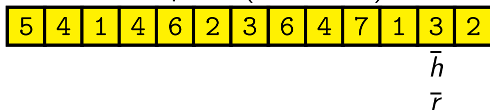
Bounded queue: example

$\bar{h} = h \bmod n$, $\bar{r} = r \bmod n$ always $h \leq r \leq h + n$

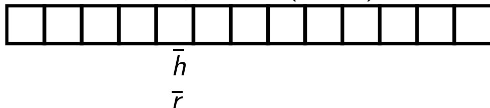




full queue ($r = h + n$)



empty queue ($r = h$)



Bounded queue: pseudocode

Queue implementation with an array $A[0..n-1]$

var $h \leftarrow 0, r \leftarrow 0$: natural

enqueue(d : datum)

    if $r = h + n$ **then**
        throw *FullQueueException*
    else
        $A[r \bmod n] \leftarrow d$
        $r \leftarrow r + 1$

dequeue()

    if $r = h$ **then**
        throw *EmptyQueueException*
    else
        $h \leftarrow h + 1$
        return $A[(h-1) \bmod n]$

Stack/queue implementation using lists

Drawbacks of an *array* implementation of a stack or queue:

- Memory is wasted if the size of the stack/queue is less than n.
- The size of the stack/queue must be bounded.

An alternative is to store each element in a node in a list structure.

For example, in case of a stack:

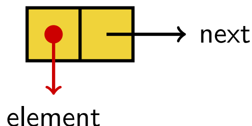
- *top* points to the top of the stack.
- If the stack is empty, *top* is a `null` pointer.
- Each node carries a pointer to its predecessor on the stack (or a `null` pointer, for the bottom element of the stack).

Singly-linked lists

A node v in a (singly-linked) list contains:

- a data element $v.element$, and
- a pointer $v.next$ to the next node in the list.

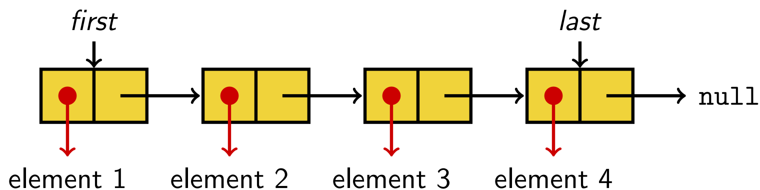
  In the last node of the list this pointer is `null`.



Singly-linked lists

Variables *first* and *last* point to the first and last node in the list.

Initially they have the value `null`.



List as an abstract data type

$\mathit{first}()$

$\mathit{last}()$

$\mathit{size}()$

$\mathit{isEmpty}()$

$\mathit{inList}(v)$

$\mathit{before}(v)$

$\mathit{after}(v)$

$\mathit{isFirst}(v)$

$\mathit{isLast}(v)$

$\mathit{insert}(d)$

$\mathit{remove}(v)$

$\mathit{replaceElement}(v, d)$

$\mathit{insertBefore}(v, d)$

$\mathit{insertAfter}(v, d)$

Question

What is the worst-case time complexity of (a naive implementation of) $inList(v)$?

Answer: $O(n)$, with n the length of the list.

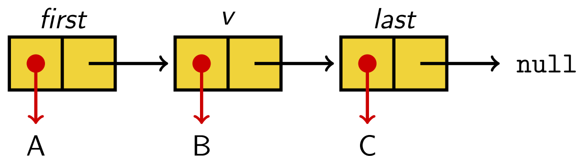
$insertAfter(v, d)$: pseudocode

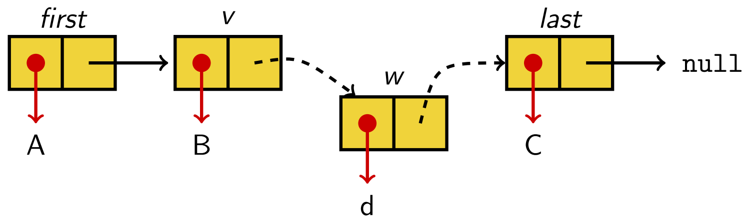
Insert a new node after a given node in a list

$insertAfter(v : \text{node}, d : \text{datum})$

if $inList(v)$ *returns false* **then**
  throw *NodeAbsentException*
new w : node
$w.element \leftarrow d$
$w.next \leftarrow v.next$
$v.next \leftarrow w$
if $last = v$ **then**
  $last \leftarrow w$

insertAfter(v, d): example





Questions

Could $w.element \leftarrow d$ be moved to the end of the program ?

Could $w.next \leftarrow v.next$ be moved to the end of the program ?

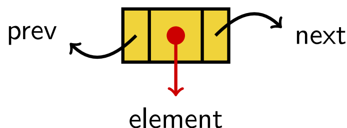
What is the worst-case time complexity of $insertAfter(v, d)$?

Answer: $O(n)$, due to the application of $inList(v)$.

Doubly-linked lists

A node v in a doubly-linked list contains:

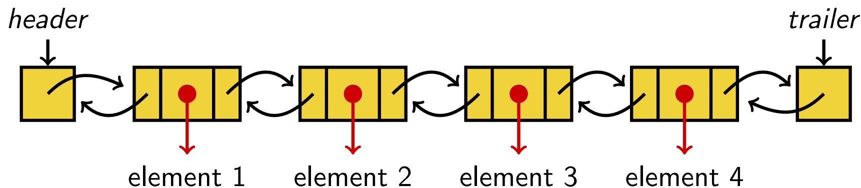
- a data element $v.element$,
- a pointer $v.next$ to the next node in the list, and
- a pointer $v.prev$ to the previous node in the list.



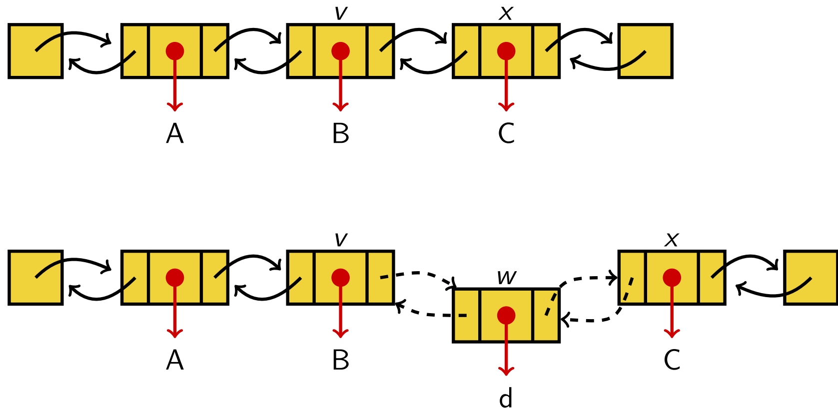
Doubly-linked lists

Variables *header* and *trailer* point to sentinel nodes, marking the start and end of the list.

These sentinel nodes initially have the value `null`.



insertAfter(v, d) in a doubly-linked list



Sorting

Given a *totally ordered* set of elements.

Input: A list of elements.

Output: The ordered permutation of this list.

In many applications, elements are actually keys of some item, and the aim is to find such items quickly.

Think of a phone book, where names are ordered alphabetically, so that corresponding phone numbers can be found quickly.

Sorting

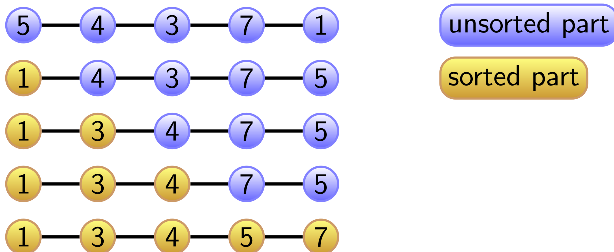
We discuss a wide range of sorting algorithms for several reasons.

- Sorting is a fundamental and important algorithmic challenge.
- Different algorithms have different (dis)advantages.
- Exemplify different algorithmic approaches.
- Show how the algorithmic design influences time complexity.
- Show the importance of tailor-made data structures.

Selection sort

While the unsorted part of the list contains more than one element:

- search for a smallest element in the unsorted part of the list,
- swap this element with the first element in the unsorted part of the list, and
- exclude this element from the unsorted part of the list.



Selection sort: pseudocode

Selection sort for an array $A[1..n]$ of naturals

var min, x : natural

for $j = 1$ **to** $n - 1$ **do**

 $min \leftarrow j$

 for $k = j + 1$ **to** n **do**

  if $A[k] < A[min]$ **then**

   $min \leftarrow k$

 $x \leftarrow A[min]$

 $A[min] \leftarrow A[j]$

 $A[j] \leftarrow x$

Question

Why do we use a help variable x to swap the values of $A[j]$ and $A[min]$?

From now on we assume a function $swap(_,_)$ to perform such swaps.

Selection sort: time complexity

The worst-case time complexity of sorting a list of length n is

$$O(n^2).$$

Most time is spent on repeatedly finding a smallest element in the unsorted part.

Each run of the outer **for** loop takes $O(n - j)$ time.

And $O((n-1) + \cdots + 1) \ = \ O(n^2)$.

Questions

What is the *best*-case time complexity of selection sort ?

Answer: $O(n^2)$

What is the *average*-case time complexity of selection sort ?

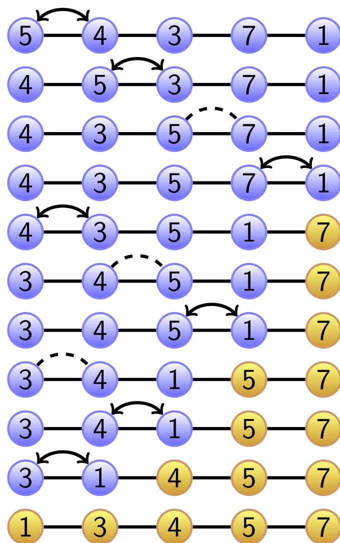
Answer: $O(n^2)$

Bubble sort

Repeatedly traverse the list from left to right, compare each pair of adjacent numbers, and swap them if they are in the wrong order.

After a traversal, the rightmost element is largest in the traversed list, so is excluded from the next traversal.

Bubble sort: example



unsorted part

sorted part

Bubble sort: pseudocode

Bubble sort for an array $A[1..n]$ of naturals

for $i = n - 1$ **downto** 1 **do**
  for $j = 1$ **to** i **do**
    if $A[j] > A[j+1]$ **then**
      $swap(A[j], A[j+1])$

Mergesort

While the list contains more than one element:

- split it into two halves of (almost) equal length,
- sort each of the two sublists using mergesort, and
- merge the two sorted sublists.

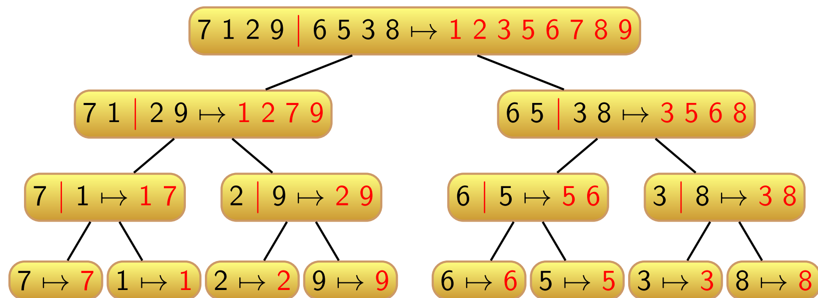
Merging two sorted lists is performed as follows.

As long as both lists are non-empty:

- compare the first element of both lists, and
- move the smallest of the two to the first empty slot in the merged list.

When one of the lists becomes empty,
add (the remainder of) the other list at the tail of the merged list.

Mergesort: example



Each node represents a recursive call.

Divide and conquer

Mergesort is a divide-and-conquer algorithm:

- Divide: split the original problem into smaller subproblems.
- Solve the subproblems using recursion.
- Conquer: combine the solutions of the subproblems into a solution of the original problem.

Logarithms

Divide-and-conquer algorithms typically have a logarithm in their time complexity, as dividing a problem of input size 2^k into subproblems of size 1 takes k steps. And $k = \log_2 2^k$.

operation	*inverse*
$m + n$	$(m+n) - n = m$
$m + n$	$(m+n) - m = n$
$m \cdot n$	$(m \cdot n)/n = m$
$m \cdot n$	$(m \cdot n)/m = n$
m^n	$\sqrt[n]{m^n} = m$
m^n	$\log_m m^n = n$

Logarithms

$\log_a n \in O(\log_b n)$ for all $a, b > 0$.

$$\log_a a^m = m \quad \Rightarrow \quad a^{\log_a a^m} = a^m \quad \Rightarrow \quad a^{\log_a n} = n.$$

$$a^{\log_a b \cdot \log_b n} = (a^{\log_a b})^{\log_b n} = b^{\log_b n} = n = a^{\log_a n}$$

Hence, $\log_a b \cdot \log_b n = \log_a n$.

So we write $O(\log n)$ instead of $O(\log_a n)$.

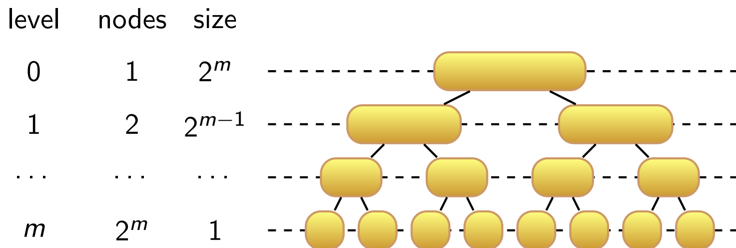
Mergesort: time complexity

Merging two sorted lists of length k and ℓ takes at most $O(k+\ell)$ time.

Splits, so also merges, take place at $\lceil \log_2 n \rceil$ levels.

In total, this takes at most $O(n)$ time per level.

So the worst-case time complexity is $O(n \cdot \log n)$.



Dynamic programming

Dynamic programming: divide a problem into subproblems.

Use *memoization*, i.e., store solutions to subproblems, to look them up when needed.

Question: The Fibonacci numbers are defined by $F(0) = F(1) = 1$ and $F(n+2) = F(n) + F(n+1)$.

Why is memoization is essential to efficiently compute these numbers ?

0-1 knapsack problem: dynamic programming solution

Given n items $i_1, \ldots, i_n$; item i_k has *value* $v_k \in \mathbb{R}$ and *weight* $w_k \in \mathbb{N}$.

Find items of total weight $\leq W$ that have a maximal total value.

$m[k, w]$ represents a solution if only the first k items can be selected and the total weight must be $\leq w$.

$$m[0, w] = 0 \qquad \text{for all } w$$

$$m[k+1, w] = \begin{cases} m[k, w] & \text{if } w_{k+1} > w \\ \max\{\, m[k, w],\ m[k, w - w_{k+1}] + v_{k+1} \,\} & \text{if } w_{k+1} \leq w \end{cases}$$

The solution to the 0-1 knapsack problem is computed in $m[n, W]$.

0-1 knapsack problem: dynamic programming solution

for $w = 0$ **to** W **do**

$\quad m[0, w] \leftarrow 0$

for $k = 0$ **to** $n - 1$ **do**

$\quad$ **for** $w = 0$ **to** W **do**

$\quad\quad$ **if** $w_{k+1} > w$ **then**

$\quad\quad\quad m[k+1, w] \leftarrow m[k, w]$

$\quad\quad$ **else**

$\quad\quad\quad m[k+1, w] \leftarrow \max\{\, m[k, w],\ m[k, w - w_{k+1}] + v_{k+1} \,\}$

0-1 knapsack problem: NP-completeness

Time complexity: $O(n \cdot W)$

This looks polynomial, but is *exponential* in the input size.

Because W is represented by $\log_{10} W$ decimals.

The 0-1 knapsack problem reformulated as a *decision problem*:

Can a total value $\geq V$ (for some given V) be achieved?

This problem is NP-complete.

It suggests that no efficient general solution exists.

Mergesort: drawback

Weak point is that merging two sorted sublists must take place outside the original list.

This is going to be solved by the following sorting algorithm.

Quicksort

While the list contains more than one element:

- Pick a number p from the list, called the pivot.
- Concurrently walk through the list from the left and from the right, until these walks meet.

  If from the *left* a number $\geq p$ and from the *right* a number $\leq p$ are encountered, then *swap* these numbers.
- Split the list into two parts, containing numbers $\leq p$ and $\geq p$, respectively.

  Recursively apply quicksort to both sublists.

Often the pivot is simply taken to be the first element of the list.

Quicksort: pseudocode

Quicksort for an array $A[k..\ell]$ of naturals

QuickSort$(A[k..\ell])$

var m : natural

if $k < \ell$ **then**

$m \leftarrow Partition(A[k..\ell])$

$QuickSort(A[k..m])$

$QuickSort(A[m+1..\ell])$

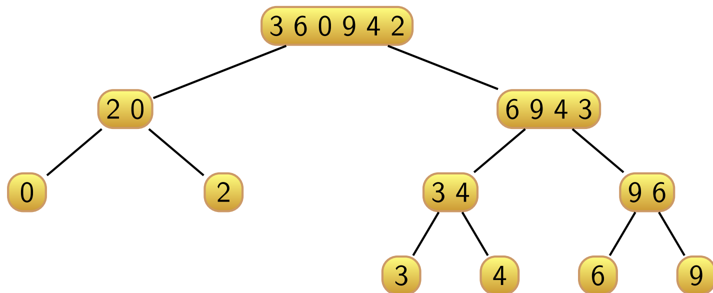
Quicksort: pseudocode

Partition$(A[k..\ell])$

var $i \leftarrow k-1, \quad j \leftarrow \ell+1, \quad pivot \leftarrow A[k]$: natural

```
while i < j do
    i ← i + 1
    while A[i] < pivot do
        i ← i + 1
    j ← j − 1
    while A[j] > pivot do
        j ← j − 1
    if i < j then
        swap(A[i], A[j])
return j
```

Quicksort: example



Each node represents a recursive call.

Quicksort: time complexity

In the worst case, quicksort takes $O(n^2)$ time to sort a list of length n.

For example, apply quicksort to the list $[1, 2, \ldots, n]$.

(Let the pivot always be the first element of the list.)

First, the original list is split into $[1]$ and $[2, \ldots, n]$.

Next, $[2, \ldots, n]$ is split into $[2]$ and $[3, \ldots, n]$. And so on.

The average-case time complexity of quicksort is $O(n \cdot \log n)$!

(In contrast to insertion, selection and bubble sort.)

Randomization of pivot selection can reduce average running time.

Questions

Why don't we swap if from the left a number $> p$ and from the right a number $< p$ are encountered ?

Answer: Else in *Partition*$(A[k..\ell])$ an additional check would be needed to ensure that indices i and j don't go outside the range $[k..\ell]$.

Why is it convenient to pick an element in the list as pivot ?

Answer: Again, else i or j could go outside the range $[k..\ell]$.

Questions

Can $QuickSort(L)$ lead to subcalls $QuickSort(L)$ and $QuickSort(\emptyset)$?

Suppose we would take the *last* element in the array as pivot.

Could then $QuickSort(L)$ lead to subcalls $QuickSort(L)$ and $QuickSort(\emptyset)$?

Quicksort: correctness

Let $k < \ell$. Suppose *Partition*$(A[k..\ell])$ returns m.

- $k \leq m < \ell$.
- *Partition* yields a permutation of the original list, where $A[k..m]$ contains only numbers $\leq A[k]$ and $A[m+1..\ell]$ contains only numbers $\geq A[k]$.

The second claim holds because i traversed $A[k..m-1]$, j halted at $A[m]$, and j traversed $A[m+1..\ell]$.

By induction, *QuickSort*$(A[k..m])$ and *QuickSort*$(A[m+1..\ell])$ produce ordered versions of $A[k..m]$ and $A[m+1..\ell]$, respectively.

These facts together imply that *QuickSort*$(A[k..\ell])$ produces an ordered version of $A[k..\ell]$.

Trees

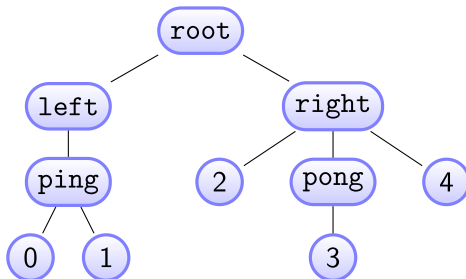
A tree consists of nodes connected by an *acyclic* parent-child relation.

There is a unique root node.

Each non-root has a unique parent.

A leaf is a node without children.

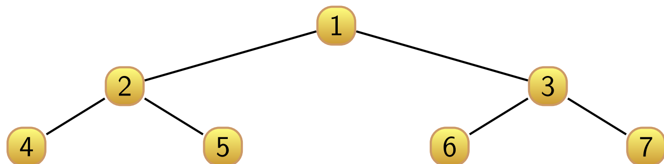
Example:



Binary trees

In a binary tree, each node has at most two children.

The root carries index 1, and if a node has index i,
then its children (if any) are indexed $2i$ and $2i + 1$.



These indices correspond to slots in an array,
and shouldn't be confused with the numbers stored in the nodes.

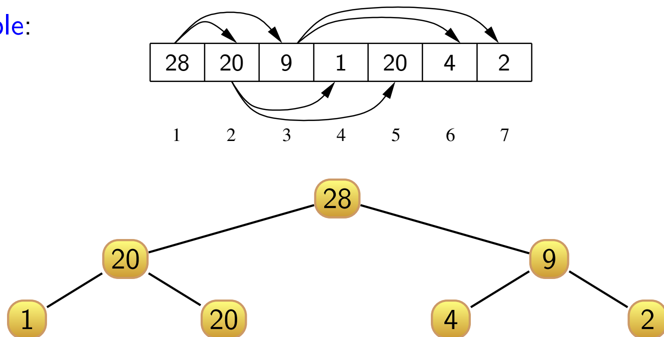
An array of length n gives rise to a binary tree of depth $\lfloor \log_2 n \rfloor$.

Heaps

A binary tree in which each node contains an element from a totally ordered set S is a heap if:

- The binary tree is completely filled at all levels, except possibly the lowest level, which is filled from left to right.
- On each path from the root to a leaf, numbers are non-increasing.

Example:



Strong points of a heap

Turning an array of length n into a heap takes $\Theta(n)$.

Finding the largest element in a heap takes $\Theta(1)$.

Replacing the element in the root of a heap, and restoring the heap, takes at most $O(\log n)$.

Question: How can we use heaps for an efficient sorting algorithm ?

Constructing a heap

The binary tree representation of an array is turned into a heap in a bottom-up fashion.

For $i = \lfloor \frac{n}{2} \rfloor$ down to 1, turn the binary tree rooted in node i into a heap as follows:

- The binary trees rooted in its children $2i$ and $2i + 1$ (if present) have already been turned into heaps.

  Let the roots of these heaps contain numbers k and ℓ.

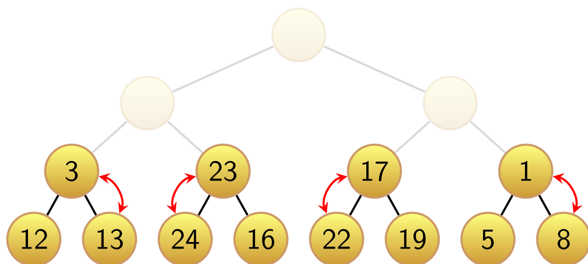
- Move the number m in node i downward, to its proper place in the heap.

  That is, if k or ℓ is larger than m, then swap m with $\max\{k, \ell\}$ and repeat this process at the lower level.

Constructing a heap: example

We build a heap from the following array of $2^4 - 1 = 15$ numbers:

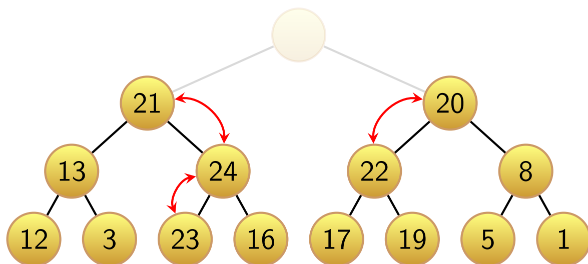
18 21 20 3 23 17 1 12 13 24 16 22 19 5 8



Constructing a heap: example

We build a heap from the following array of $2^4 - 1 = 15$ numbers:

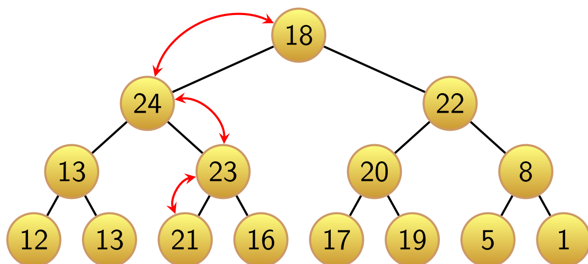
18 21 20 3 23 17 1 12 13 24 16 22 19 5 8



Constructing a heap: example

We build a heap from the following array of $2^4 - 1 = 15$ numbers:

18 21 20 3 23 17 1 12 13 24 16 22 19 5 8



Constructing a heap: pseudocode

Turning $A[1..n]$ into a heap

Heapify$(A[1..n])$

var *start* : natural

```
start ← ⌊n/2⌋
while start > 0 do
    SiftDown(A[start..n])
    start ← start − 1
```

SiftDown$(A[k..\ell])$ assumes $A[k+1..\ell]$ has the proper heap structure, and (recursively) sifts $A[k]$ down to a proper place in the heap.

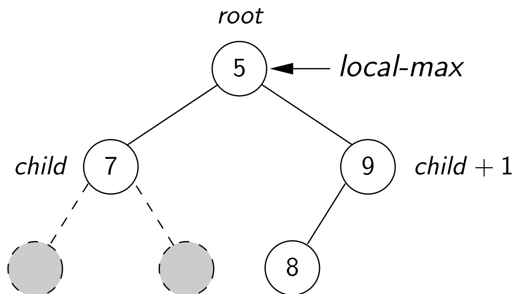
Constructing a heap: pseudocode

SiftDown$(A[k..\ell])$

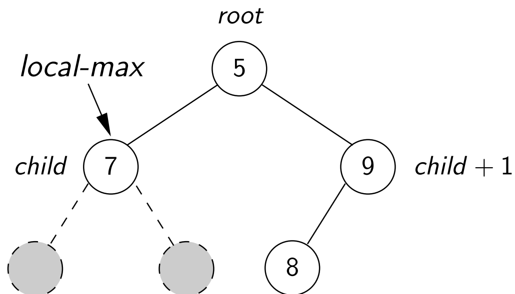
var $\mathit{root}, \mathit{child}, \mathit{local\text{-}max}$: natural

```
root ← k
local-max ← k
while 2·root ≤ ℓ do
    child ← 2·root
    if A[local-max] < A[child] then
        local-max ← child
    if child + 1 ≤ ℓ and A[local-max] < A[child + 1] then
        local-max ← child + 1
    if local-max ≠ root then
        swap(A[root], A[local-max])
        root ← local-max
    else
        return
```

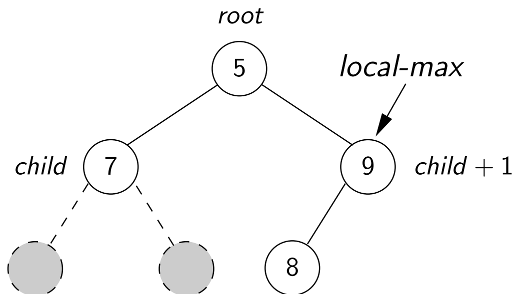
Local comparisons: example



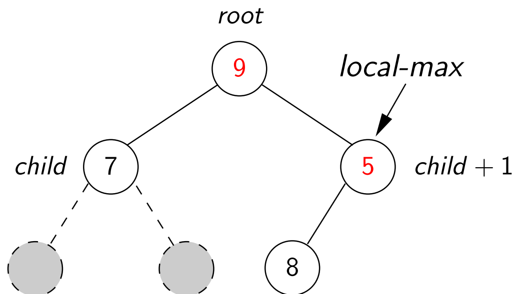
Local comparisons: example



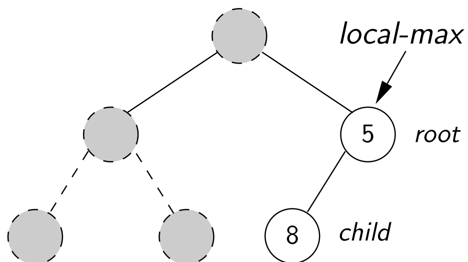
Local comparisons: example



Local comparisons: example



Local comparisons: example



Constructing a heap: time complexity

Turning an array of n numbers into a heap takes

$$\Theta(n).$$

The total number of moves to sift down numbers is smaller than the number of edges in the binary tree (i.e., $n - 1$).

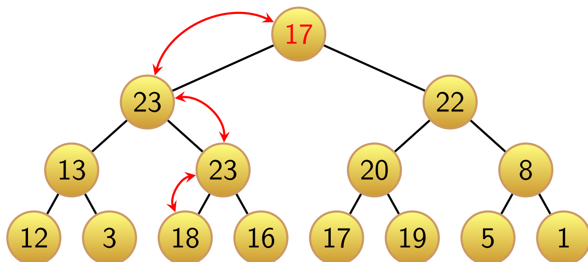
This can be seen by depicting a binary graph:

Even if each move is over a different edge, there is always an edge on which no move is performed.

Reconstructing a heap

If the number in the root (the maximum) is replaced by another number, *SiftDown* can be used to reconstruct the heap.

Example:



Reconstructing a heap: time complexity

Reconstructing a heap of n nodes after the number in the root has been changed takes at most

$$O(\log n).$$

In the worst case, the edges on a path from the root to a leaf are taken into account.

And a heap of depth k can contain up to $2^{k+1} - 1$ nodes.

Heapsort

First structure the array of numbers into a heap.

While the heap contains more than one node:

- swap the number in the root and the number in the last node of the heap;
- exclude the last node from the heap; and
- reconstruct the heap.

Heapsort: pseudocode

Heapsort for an array $A[1..n]$

var *end* : natural

```
Heapify(A[1..n])
end ← n
while end > 1 do
    swap(A[1], A[end])
    end ← end − 1
    SiftDown(A[1..end])
```

Heapsort: time complexity

Building the heap takes $\Theta(n)$ time.

Reconstructing the heap takes at most $O(\log n)$ time.

This is done at most $n - 2$ times.

So the overall worst-case time complexity is $O(n \cdot \log n)$.

Sorting based on pairwise comparisons

Theorem: Sorting algorithm based on pairwise comparison of numbers in the list have worst-case time complexity at best $O(n \cdot \log n)$.

Proof: Draw a binary tree of all possible steps a sorting algorithm based on pairwise comparison can make.

Each node in the tree is an intermediate state of the algorithm while being applied to an (unsorted) list of n distinct numbers.

The root of the tree represents an input list; each leaf represents that the algorithm terminates with a sorted list.

From each non-leaf, if the next comparison yields *true* we go left in the tree, and if the next comparison yields *false* we go right.

There are $n!$ ways to put the n numbers in a list, and each of these lists takes a different path in the tree. So there are $n!$ leaves.

This implies the tree has depth at least $\log_2 n! \ = \ n \cdot \log n$.

So in the worst case the algorithm takes at least $O(n \cdot \log n)$.

Counting sort

Assumption: The set of elements has size m.

Counting sort counts how many times each element occurs in the list.

The time complexity is $\Theta(m + n)$:

$\Theta(n)$ for counting and $\Theta(m)$ for turning the counts into a sorted list.

Drawback: Requires an additional array of length m.

m can be very large (for instance in case of a phone book).

Radix sort

Assumption: Numbers in the list contain at most k digits.

Radix sort sorts the list on each of the k digits consecutively, starting with the *least significant digit*.

Bit-wise sorting is performed using so-called *bucket sort*, where numbers are placed in 10 "buckets".

1. Sort numbers in the list based on the least significant digit, but otherwise keep the original order of numbers.

   Numbers with the same least significant digit are stored in the same bucket.

2. Repeat this process with each more significant digit.

The time complexity is $\Theta(k \cdot n)$.

Question

Sort the list below using radix sort:

170 45 75 90 802 24 2 66

Searching

Input: An *ordered* array of distinct keys, and a given key.

Output: The position in the array where the key can be found,
or a negative reply that the key isn't present in the array.

Example: A phone book, where names are ordered alphabetically,
so that phone numbers can be found quickly.

Binary search

The binary search algorithm compares the searched key value with the key value in the middle of the array (while it is non-empty).

If the keys *match*, then return the corresponding index of the array.

If the searched key is *smaller* than the middle element's key, then the search is repeated on the sub-array to the *left* of the middle element.

If the searched key is *greater* than the middle element's key, then the search is repeated on the sub-array to the *right* of the middle element.

If the array to be searched is empty, then the searched key isn't present in the original array.

Binary search

Search for a key k in a sorted array $A[\ell..m]$ as follows.

If $\ell \leq m$, then compare $A[n]$ and k, with $n = \lfloor \frac{\ell+m}{2} \rfloor$.

- If $A[n] = k$, then return that k is at slot n in the array.

  If $A[n] > k$, then continue the binary search with the array $A[\ell..n-1]$.

  If $A[n] < k$, then continue the binary search with the array $A[n+1..m]$.

If $\ell > m$, then return that k isn't present in the array.

Binary search: time complexity

To determine whether (and where) a key can be found in a sorted array of length n takes at most

$$O(\log n).$$

Because each comparison takes $O(1)$, and at the subsequent recursive call, the length of the sub-array to be searched is halved.

Sorted arrays are however not suited to dynamic sets of keys.

- Adding or deleting an element may take $O(n)$.
- The maximal size of the set must be predetermined.

Binary search trees

A binary search tree is a binary tree in which nodes contain a key, such that for every node ν:

- the *left* subtree of ν contains only nodes with keys *smaller* than ν's key; and
- the *right* subtree of ν contains only nodes with keys *greater* than ν's key.

Searching for a key can be performed in $O(d)$, with d the depth of the tree.

We search for key k in a binary search tree.

If the binary search tree is non-empty,
then compare k with the key ℓ in the root of the tree.

- $k = \ell$: report that the key has been found.
- $k < \ell$: (recursively) search for k in the *left* subtree of the root.
- $k > \ell$: (recursively) search for k in the *right* subtree of the root.

If the tree is empty, then report that k isn't present (in the original tree).

Adding a key to a binary search tree

We add a key k to a binary search tree.

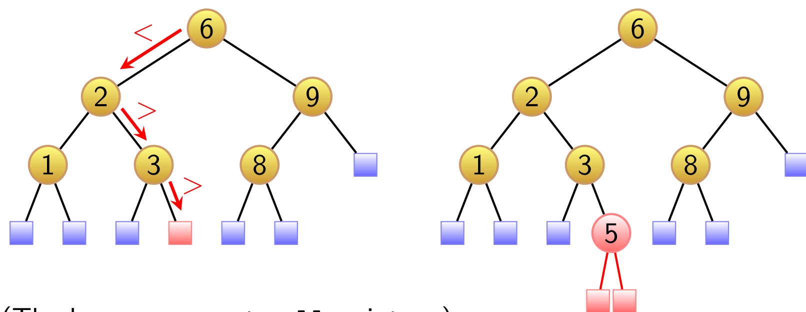
If the binary search tree is non-empty,
then compare k with the key ℓ in the root of the tree.

- $k = \ell$: report that k is already present in the tree.
- $k < \ell$: (recursively) add k in the *left* subtree of the root.
- $k > \ell$: (recursively) add k in the *right* subtree of the root.

If the tree is empty, then create a (root) node containing k.

Adding a key to a binary search tree: example

We add key 5 to the binary search tree below.



(The boxes represent `null` pointers.)

Question: Add key 4 to the binary search tree at the right.

Question: Which binary search tree results if first 4 and then 5 is added to the binary search tree at the left?

Removing a key from a binary search tree

A key k is removed from a binary search tree as follows.

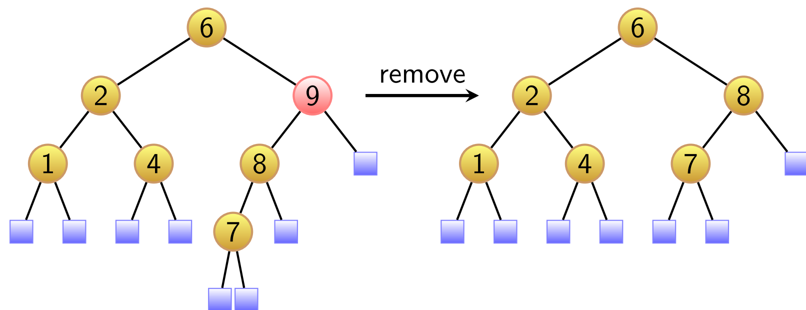
Search for k in the tree.

- If k is not present in the tree, we are done.
- If k is in a leaf, then remove this leaf from the tree.
- If k is in a node ν with one child, then replace ν by its child.
- If k is in a node ν with two children, then:
  - search for the node ν' holding the *smallest* key ℓ in the *right* subtree of ν,
  - replace k by ℓ in ν, and
  - (recursively) remove ν' from the tree.

Removing a key from a binary search tree: example 1

We remove the node with key 9 from the binary search tree below.

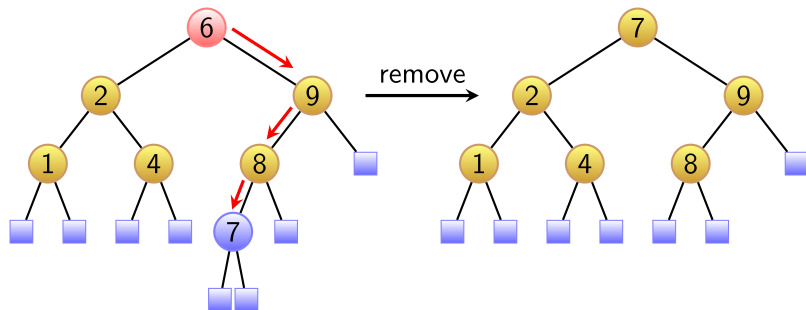
Note that this node has *one* child.



Removing a key from a binary search tree: example 2

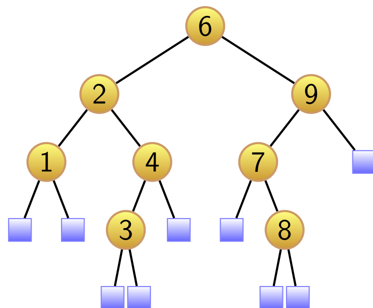
We remove the node with key 6 from the binary search tree below.

Note that this node has *two* children.



Questions

Question: Remove 6 from the following binary search tree.



Question: When removing a node ν with two children, we search for the node holding the *smallest* key in the *right* subtree of ν.

Could we alternatively search for the node holding the *largest* key in the *left* subtree of ν?

Adapting a binary search tree: time complexity

Adding or removing a key from a binary search tree both take at most $O(d)$, with d the depth of the tree.

Because in the worst case one path from the root to a leaf is traversed.

Question: In which type of binary search trees may a search take relatively long ?

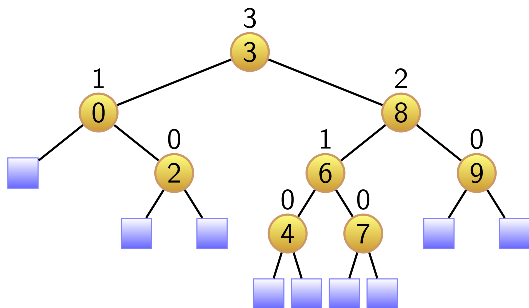
AVL trees

Binary search trees are efficient if their depth is relatively small.

A binary search tree is an AVL tree if for any node, the depths of the two subtrees differ at most one.

An AVL tree of n nodes has depth $O(\log n)$.

AVL trees: example



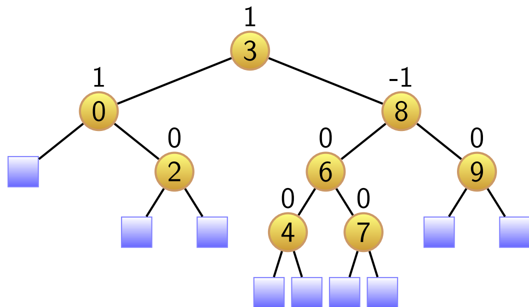
At each node, the depth of the subtree rooted at this node is given.

AVL trees: balance factor

The balance factor of a node in a binary tree is
the depth of its right subtree minus the depth of its left subtree.

In an AVL tree, the balance factor of any node is -1, 0, or 1.

Example:



Challenge: After adding or removing a node, rebalance the tree (if needed) to preserve the defining property of AVL trees.

Adding a key to an AVL tree

A key is added to an AVL tree in the same way as in a binary search tree.

As a result, the balance factor of nodes on the path from the root to the added node may change.

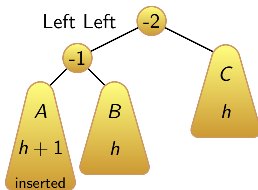
From the added node upward, check if a node has balance factor -2 or 2.

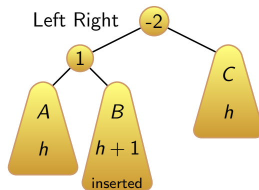
If such a node is found, perform a (single or double) rotation.

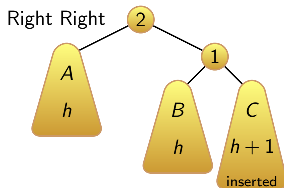
After this local operation, the tree is guaranteed to be an AVL tree again.

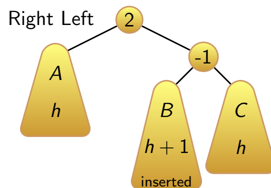
Adding a key to an AVL tree

Four cases for the *lowest* node with balance factor -2 or 2.









Adding a key to an AVL tree: Left Left

In the Left Left case, the depth of tree A increases from h to $h+1$.

Since this increases the depth of tree A-B, tree B has a depth $\leq h$.

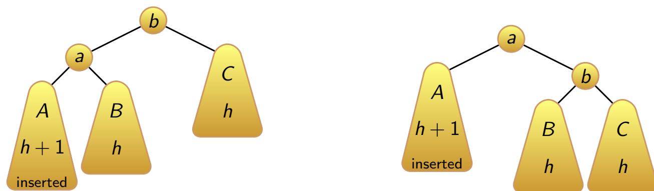
Since tree A-B is still balanced after addition, tree B has depth h.

Since tree A-B-C was balanced before addition, tree C has depth $\geq h$.

Since tree A-B-C is imbalanced after addition, tree C has depth h.

Adding a key to an AVL tree: single rotation

In the Left Left case, a single rotation is applied:



The result is still a binary search tree, in which all nodes are balanced.

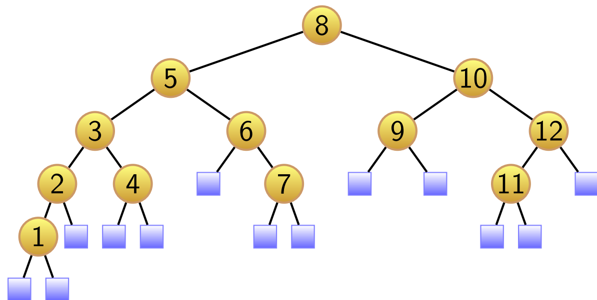
The single rotation restores the depth $h+1$ of the original subtree.

Therefore, after this single rotation, the *entire* tree is an AVL tree again.

Likewise, in the Right Right case, a single rotation is applied.

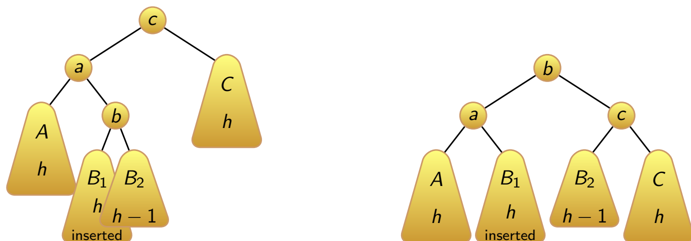
Question

Add a node with key 0 to the AVL tree below, and restructure the resulting tree.



Adding a key to an AVL tree: double rotation

In the Left Right case, a double rotation is applied:



The result is still a binary search tree, in which all nodes are balanced.

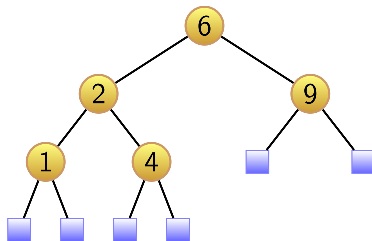
The double rotation restores the depth $h + 1$ of the original subtree.
Therefore, after this double rotation, the *entire* tree is an AVL tree again.

Likewise, if the added node is in B_2 or is b, a double rotation is applied.

Also in the Right Left case, a double rotation is applied.

Questions

How is 3 added to the AVL tree below ?



And how is 5 added to the AVL tree above ?

Adding a key to an AVL tree: correctness

Only nodes on the path from the root to the added node may become imbalanced.

The rotation performed at the lowest imbalanced node, say ν, makes sure the subtree rooted in ν becomes an AVL tree again.

After rotation, the subtree rooted in ν has the same depth as before addition.

These two facts imply that after rotation, the entire tree is AVL.

Removing a key from an AVL tree

A key is removed from an AVL tree in the same way as from a binary search tree.

As a result, the balance factor of nodes on the path from the root to the removed node may change.

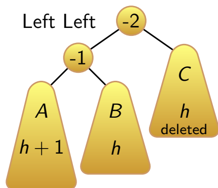
From the removed node upto the root, we need to possibly check *for every node* on this path if it has balance factor -2 or 2.

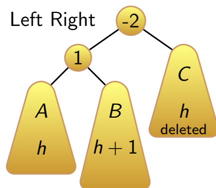
Every time such a node is found, perform a (single or double) rotation.

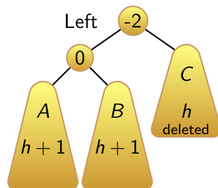
The resulting tree is guaranteed to be an AVL tree again.

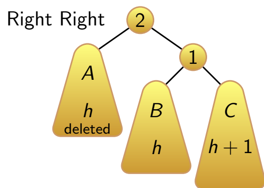
Removing a key from an AVL tree

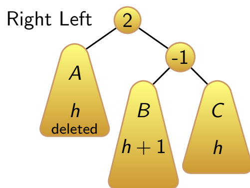
Six cases for a node with balance factor -2 or 2.

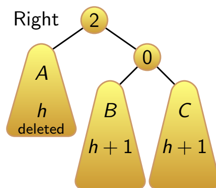












Removing a key from an AVL tree

The Left Left, Right Right, Left Right, and Right Left case are treated as before.

Question: Show that in the Left Left (and Right Right) case, the single rotation affects the depth of the original subtree.

Likewise, in the Left Right (and Right Left) case, the double rotation affects the depth of the original subtree.

Removing a key from an AVL tree

The Left case is similar to the Left Left case.

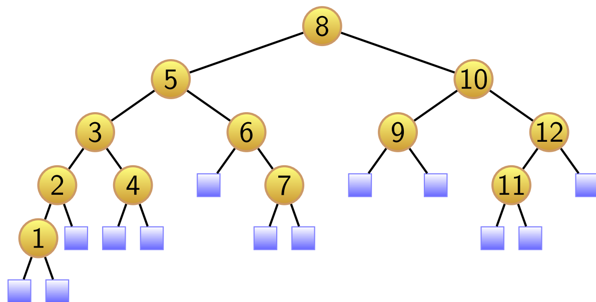
A single rotation is applied:

Likewise, in the Right case, a single rotation is applied.

In the Left and Right case, the depth of the original subtree is kept.

Question

Remove the node with key 9 from the AVL tree below.



Note that after the double rotation for this Right Left case (with $h = 0$) at node 10, the root 8 becomes imbalanced.

Adapting an AVL tree: time complexity

Adding a node to or removing a node from an AVL tree with n nodes takes at most

$$O(\log n).$$

Because in the worst case, a path from a leaf to the root is traversed.

And a (single or double) rotation takes $O(1)$.

Direct-address table

Let there be m possible keys.

A direct-address table uses an array of size m: one slot per key.

Slot k points to the element with key k (if present).

Advantages: Inserting and deleting an element takes $O(1)$, and searching for a key is straightforward.

Drawback: Ineffective if only few keys are actually used.

Hash function

A hash function h maps keys to slots $\{0, \ldots, m-1\}$ in a hash table.

The number of keys is considerably larger than m.

Hashing

- casts data from a large (sparsely used) domain to a fixed-size table, and
- allows to perform fast lookups.

Collisions occur: different keys are hashed to the same value.

Hashing: applications in cryptography

Hashing is also used in *cryptography*, such as:

- digital signatures of messages, or
- storing passwords safely.

For a cryptographic hash function:

- hash values are easy to compute, but
- it is hard find a key that produces a given hash value.

In general, the last property is very hard to check.

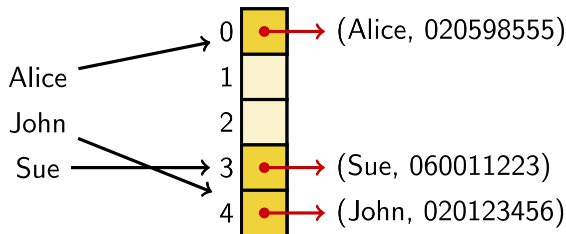
For example, it took several years to discover that hash functions MD5 and SHA-1 aren't *collision resistant*.

Hashing: example

We only consider simplistic hash functions, like modulo, to exemplify the underlying idea.

Let a hash function map names to numbers from 0 to 4.

Elements are phone numbers.



Collisions

A collision for a hash function h occurs if elements for different keys k, k' are stored in the hash table and $h(k) = h(k')$.

There are two ways to deal with collisions:

- Chaining: Put keys that hash to the same value j in a linked list.

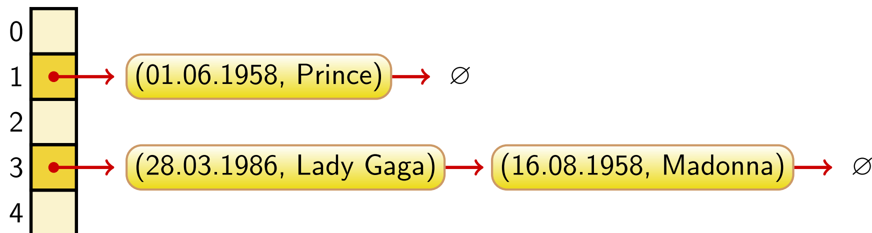
  Slot j in the hash table points to the head of this list.

- Open addressing: If a key isn't found in the designated slot in the hash table, a probe function tells which next slot to try.

  Each non-empty slot contains an element; there are no pointers.

Chaining: example

The hash function is the month of birth modulo 5.



Disadvantage: Pointer structures are expensive to maintain and traverse.

Open addressing: linear probing

A simple approach to open addressing is linear probing.

If a wrong key is found in a slot, then try the next slot (modulo m).

If an element is removed from a table, a marker is placed in its slot, to signify that linear probing should continue onward.

If a slot is found to be empty and unmarked, give up the search.

Disadvantage: Long sequences of occupied/marked slots tend to build up, increasing the average search time.

Open addressing: double hashing

Introduce an auxiliary hash function h' (that never hashes to 0).

Searching for a key k is performed as follows:

If a wrong key is found in slot j, then try slot $j + h'(k)$, modulo m.

Example: $m = 13$, $h(k) = k \bmod 13$, and $h'(k) = 8 - (k \bmod 7)$.

k	$h(k)$	$h'(k)$	try
18	5	4	5
44	5	6	5, 11
59	7	5	7
32	6	4	6
72	7	6	7, 0
71	6	7	6, 0, 7, 1

Hash functions: division method

Ideally, roughly the same number of keys is hashed to any of the m slots.

The division method hashes a key k to $k \bmod m$.

Advantage: Easy to compute.

Disadvantage: Doesn't work well for all values of m.

For example, when hashing a bit string, values $m = 2^\ell$ should be avoided.

Because then the hash value of k consists of the ℓ lowest-order bits of k.

Preferably the hash value of k depends on all its bits.

Hash functions: multiplication method

Choose a constant c with $0 < c < 1$.

The multiplication method hashes a key k as follows:

- compute $c \cdot k$,
- keep the fractional part of this number, $c \cdot k - \lfloor c \cdot k \rfloor$,
- multiply the result with m, and
- return the floor of this product.

In short,

$$h(k) \;=\; \lfloor m \cdot (c \cdot k - \lfloor c \cdot k \rfloor) \rfloor .$$

Hash functions: multiplication method

Example: $c = 0.1$ and $m = 17$.

We compute the hash value of $k = 137$.

- $c{\cdot}k - \lfloor c{\cdot}k \rfloor \ = \ 13.7 - 13 \ = \ 0.7$
- $m{\cdot}(c{\cdot}k - \lfloor c{\cdot}k \rfloor) \ = \ 11.9$
- $\lfloor m{\cdot}(c{\cdot}k - \lfloor c{\cdot}k \rfloor) \rfloor = 11$

Hash functions: multiplication method

Advantage: The value of m isn't critical.

It works with practically any value of c (if it isn't very close to 0 or 1), but some values are better than others.

An optimal value for c depends on the data being hashed.

Donald Knuth recommends as value for c the fractional part of the *golden ratio*

$$\frac{\sqrt{5}-1}{2} \; = \; 0.6180...$$

The golden ratio e.g. governs the placement of petals in plants. Repeated rotation over this distance creates gaps that are at most twice as big as the other gaps.

Graphs

A graph consists of nodes, and edges between a pair of nodes.

In a directed graph, edges are directed, meaning that they can only be traversed in one direction.

In an undirected graph, edges can be traversed in either direction.

Each node maintains an *adjacency list* of neighbor nodes.

Examples: hardware circuits, computer networks, TomTom, ...

Graph traversal

We consider two ways to traverse a (directed) graph.

- Breadth-first search starts from a distinguished node called the *root*.

  It places discovered unvisited nodes in a *queue*, so that nodes close to the root are visited first.

- Depth-first search places discovered unvisited nodes in a *stack*, so that recently discovered nodes are visited first.

Time complexity: $\Theta(m)$, with m the number of edges.

Breadth-first search: pseudocode

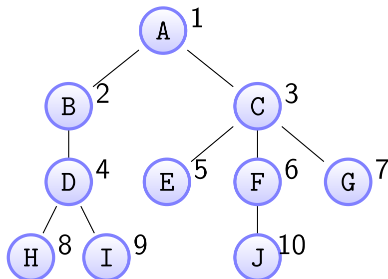
Breadth-first search through a directed graph

```
var  q ← {root} : queue of nodes
     v, w : node
     black[w : node] ← false : bool

black[root] ← true
while q ≠ ∅ do
    v ← dequeue(q)
    for all edges vw do
        if black[w] = false then
            enqueue(w, q)
            black[w] ← true
```

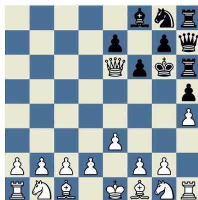
Breadth-first search: example

Breadth-first search is applied on the following undirected graph, with root node A.



Question: Add an edge from node A to node J, and perform a breadth-first search.

Question: How can breadth-first search be used to find a shortest chess game ending in checkmate (4 moves) or stalemate (19 moves) ?



Depth-first search:pseudocode

Depth-first search through a directed graph

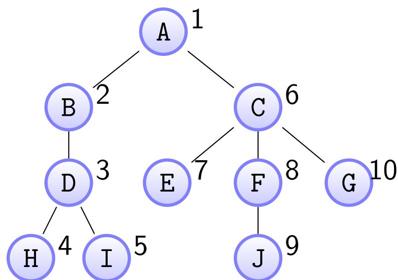
var v, w : node
  $black[w : \text{node}] \leftarrow false$: bool

for all nodes v **do**
  if $black[v] = false$ **then**
    $DFS(v)$

$DFS(v : \text{node})$

$black[v] \leftarrow true$
for all edges vw **do**
  if $black[w] = false$ **then**
    $DFS(w)$

Depth-first search: example



Question: Add an edge from node E to node J, and perform a depth-first search.

Strongly connected components

Two nodes u and v in a directed graph are strongly connected if there are paths from u to v and from v to u.

Being strongly connected is an *equivalence relation*.

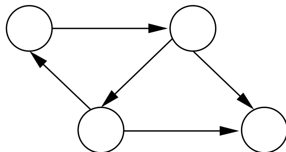
The equivalence classes are strongly connected components (SCCs).

The SCCs form a *partition* of the nodes in the graph.

Strongly connected components

SCC detection has many applications, e.g., finding cyclic garbage in memory (to support reference-counting garbage collection).

Question: What are the SCCs in the following graph ?



Depth-first search with time stamps

Let depth-first search provide "time stamps" $d[v]$ and $f[v]$ when it reaches and deserts a node v, respectively.

Depth-first search through a directed graph

var $black[w : \text{node}] \leftarrow false$: bool
    $d[w : \text{node}]$, $f[w : \text{node}]$, $time \leftarrow 1$: int

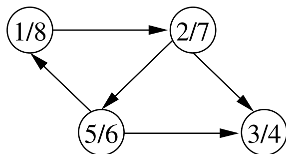
for all nodes v **do**
    if $black[v] = false$ **then**
        $DFS(v)$

Depth-first search with time stamps

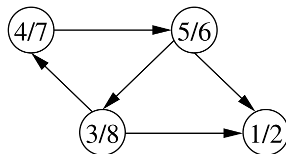
$DFS(v : \text{node})$

$black[v] \leftarrow true$
$d[v] \leftarrow time$
$time \leftarrow time + 1$
for all edges vw **do**
    if $black[w] = false$ **then**
        $DFS(w)$
$f[v] \leftarrow time$
$time \leftarrow time + 1$

Depth-first search: example



or



Kosaraju's algorithm

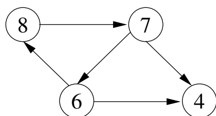
Kosaraju's algorithm for detecting SCCs in a directed graph G:

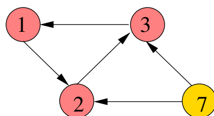
1. Apply depth-first search to G.
2. Reverse all edges in G, to obtain a directed graph G^R.

   Apply depth-first search to G^R; each new exploration starts at the unvisited node with the highest f-value.

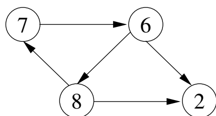
   Each exploration in G^R determines an SCC in G.

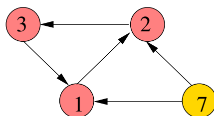
Kosaraju's algorithm: example





or





f- and d-values of the first and second depth-first search are given, respectively.

Kosaraju's algorithm: correctness

Correctness: Clearly, nodes in the same SCC are discovered during the same subcall of depth-first search on G^R.

Let node v be discovered during the subcall of depth-first search on G^R started at node u.

Then $f[u] \geq f[v]$, and there is a path from v to u in G.

Suppose, toward a contradiction, there is no path from u to v in G.

- If depth-first search on G visited v before u, then $f[v] > f[u]$ (because there is a path from v to u).
- If depth-first search on G visited u before v, then $f[v] > f[u]$ (because there is no path from u to v).

Both cases contradict the fact that $f[u] \geq f[v]$.

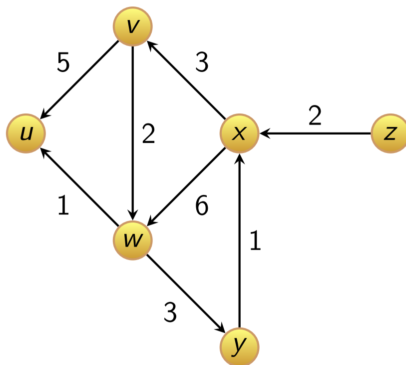
Hence v is in the same SCC as u.

Weighted graphs

In a weighted graph, each edge carries a real value.

$\omega(uv)$ denotes the weight of the edge from node u to node v.

Example:



Shortest paths in a weighted graph

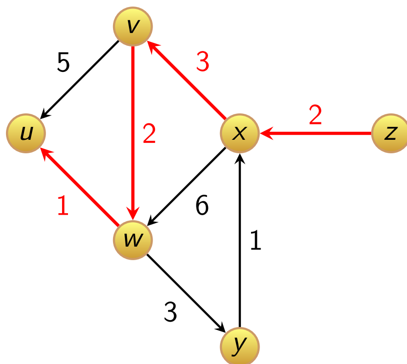
In a shortest path between two nodes in a weighted graph, the sum of the weights of the edges in the path is minimal.

Applications:

- network routing on the Internet, or for telecommunication
- TomTom
- plant layouts
- transportation

Shortest paths in a weighted graph: example

A shortest path from z to u, of weight 8.



Max- versus min-heaps

A min-heap is a binary tree in which on all paths from the root to a leaf, numbers are non-decreasing.

By contrast, in a max-heap, which we used in heapsort, on all paths from the root to a leaf, numbers are non-*increasing*.

Min-heaps can be constructed and maintained in the same way as max-heaps.

Dijkstra's shortest path algorithm

Dijkstra's algorithm computes the shortest paths from all nodes to a node u, in a directed weighted graph.

It requires that all weights are *non-negative*.

It uses relaxation: start with an *overapproximation* of distance values, and consider all edges in turn to try to improve distance values.

Initially,

- $\delta(u) = 0$;
  $\delta(v) = \infty$ for all nodes $v \neq u$;
- $\nu(v) = \perp$ for all nodes v;
- the min-heap H contains all nodes, ordered by their δ-values.

Dijkstra's algorithm

While H is non-empty:

- For each neighbor $w \in H$ of the *root* node v, check whether

  $$\delta(v) + \omega(wv) < \delta(w).$$

  If so, $\delta(w) \leftarrow \delta(v) + \omega(wv)$ and $\nu(w) \leftarrow v$, and
  w is pushed upward in the heap to its proper place.

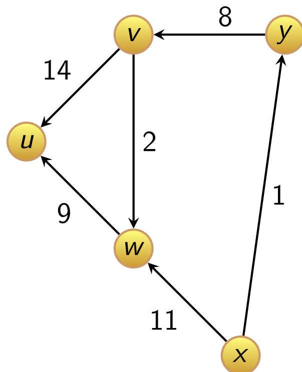
- Swap v with the last element of H, and exclude v from H.

- Restructure H into a min-heap again.

When H becomes empty, the ν-values constitute shortest paths, and δ-values the distances of shortest paths.

Question

Apply Dijkstra's algorithm to the weighted graph below, to compute shortest paths to node u.



Dijkstra's algorithm: correctness

When a node v is removed from H, $\delta(v)$ has the correct value.

We apply induction on the order in which nodes are removed from H.

The base case, $v = u$, is trivial, because initially $\delta(u) = 0$.

In the inductive case, let $v \neq u$ be the root of H.

Consider a *shortest* path $v \rightarrow^* y \rightarrow z \rightarrow^* u$, where y is the last node on this path that is still in H.

Since $z \notin H$, by induction, $\delta(z)$ and so $\delta(y)$ has the correct value.

Since v is the root of H, $\delta(v) \leq \delta(y)$.

Since weights are nonnegative, $\delta(v) = \delta(y)$, which implies that $\delta(v)$ has the correct value (and $v \rightarrow^* y$ has distance 0).

Dijkstra's algorithm: time complexity

Performing Dijkstra's algorithm on a weighted graph with n nodes and m edges takes at most

$$O(m \cdot \log n).$$

For each edge, the heap may be restructured once, taking at most $O(\log n)$.

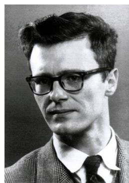
All other operations take less time:

initialize δ/ν $O(n)$, build heap $O(n)$, update δ/ν $O(m)$, restructure heap after removing a node $O(n \cdot \log n)$.

(Each node w carries a bit that is set when w is removed from H, so that the check $w \in H$ takes $O(1)$.)

Dijkstra's algorithm: inventor

Dijkstra's algorithm was invented by Edsger Dijkstra in 1956.



Dijkstra's original implementation had a worst-case time complexity of $O(n^2)$, because he didn't employ a heap.

By using a so-called Fibonacci-heap, the worst-case time complexity can actually be brought down to $O(m + n \cdot \log n)$.

Negative weights

In practice, edges in graphs may carry *negative* weights.

Example: Distance-vector routing protocols include negative costs.

Question: Give an example of a graph with one negative-weight edge on which Dijkstra's algorithm produces an incorrect result.

In case of a *negative-weight cycle*, there are no shortest paths.

Bellman-Ford algorithm

The Bellman-Ford algorithm computes the shortest paths from all nodes to a node u, in a directed weighted graph.

It allows weights to be *negative*.

Again it uses *relaxation*, but now all edges are considered n times.

Initially,

- $\delta(u) = 0$;
  $\delta(v) = \infty$ for all nodes $v \neq u$;
- $\nu(v) = \perp$ for all nodes v.

Bellman-Ford algorithm: pseudocode

We use that each shortest path contains at most $n-1$ edges.

Except when there is a negative-weight cycle !

Bellman-Ford shortest path algorithm on a weighted graph G

---

for $i = 1$ **to** $n-1$ **do**
    for each edge wv **do**
        if $\delta(v) + \omega(wv) < \delta(w)$ **then**
            $\delta(w) \leftarrow \delta(v) + \omega(wv)$
            $\nu(w) \leftarrow v$

for each edge wv **do**
    if $\delta(v) + \omega(wv) < \delta(w)$ **then**
        return "The graph contains a negative-weight cycle"

Bellman-Ford algorithm: correctness

We argue by induction: if a shortest path from v to u contains $\leq i$ edges, then after i runs of the first **for** loop, $\delta(v)$ has the correct value.

The *base* case, $i = 0$, is trivial, as $\delta(u) = 0$ at the start.

In the *inductive* case, suppose we proved the result for all $i \leq j$.

Consider a shortest path $v \rightarrow w \rightarrow^* u$ of $j + 1$ edges.

By induction, $\delta(w)$ is correct after j runs of the **for** loop.

So in run $j + 1$, $\delta(v)$ attains the correct value $\delta(w) + \omega(vw)$.

Bellman-Ford algorithm: time complexity

The time complexity of the Bellman-Ford algorithm is

$$\Theta(m \cdot n).$$

This is due to the two nested **for** loops, of n and m iterations.

Minimum spanning trees

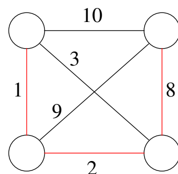
Consider an undirected, weighted graph.

A spanning tree is an acyclic connected subgraph containing all nodes.

In a minimum spanning tree, the sum of the weights of the edges in the spanning tree is *minimal*.

Applications: Design of networks for e.g. computers, telecommunication, transportation, water supply, and electricity.

Example:



Fragments

Lemma: Let F be a fragment
(i.e., a connected subgraph of a minimum spanning tree M).

Let e be a *lowest-weight* outgoing edge of F
(i.e., e has exactly one endpoint in F).

Then $F \cup \{e\}$ (i.e., F extended with e) is a fragment.

Proof: Suppose not.

$M \cup \{e\}$ has a cycle, containing e and another outgoing edge f of F.

Replacing f by e in M yields a minimum spanning tree.

Prim's and Kruskal's algorithm

Prim's algorithm is *centralized*:

- Initially, F is a *single* node.
- As long as F isn't a spanning tree,
  add a lowest-weight outgoing edge of F to F.

Kruskal's algorithm is *decentralized*:

- Initially, *each* node forms a separate fragment.
- Each step, a lowest-weight edge joining two different fragments is added to the spanning tree.

Prim's algorithm

Prim's algorithm employs *relaxation*. Initially,

- one node is selected, and provided with value 0,
- all other nodes carry value ∞, and
- the nodes are placed in a *min-heap* H, ordered by their value.

While H is non-empty:

- For each neighbor $w \in H$ of the *root* node v, check if $\omega(wv) < \mathit{value}(w)$.

  If so, $\mathit{value}(w) \leftarrow \omega(wv)$ and $\nu(w) \leftarrow v$, and
  w is pushed upward in the heap to its proper place.
- Swap v with the last element of H, and exclude v from H.
- Restructure H into a min-heap again.

Ultimately, the ν-values define a minimum spanning tree.

Prim's algorithm: time complexity

Similar to Dijkstra's algorithm, Prim's algorithm on a weighted graph with n nodes and m edges takes at most

$$O(m \cdot \log n).$$

For each edge, the heap may be restructured once, taking at most $O(\log n)$.

Prim's (and Dijkstra's) algorithm is *greedy*.

Greedy: make locally optimal choices, to find a global optimum.

NP-complete 0-1 knapsack problem: greedy approach

Given n items $i_1, \ldots, i_n$; item i_k has *value* $v_k \in \mathbb{R}$ and *weight* $w_k \in \mathbb{N}$.

Find items of total weight $\leq W$ that have a maximal total value.

Greedy: Let $\frac{v_k}{w_k}$ denote the "attractiveness" of item i_k.

Repeatedly select a most attactive item that still fits in the knapsack.

This takes polynomial time, but may yield a non-optimal solution.

Example: Given items $(3, 1)$ and $(5, 2)$ and $(6, 3)$ (the 1st and 2nd index index are the value and weight). The total weight mustn't exceed 5.

The greedy approach selects $(3, 1)$ and $(5, 2)$, while selecting $(5, 2)$ and $(6, 3)$ is a better solution.

Kruskal's algorithm

Initially,

- each node is in a single-element set (representing a fragment),
- and the edges are placed in a list, ordered by weight.

While the list of edges is non-empty, the first edge e is removed.

If e is between two nodes in different fragments, then:

- e becomes part of the minimum spanning tree, and
- these fragments are joined.

Fragments are managed using the disjoint-set data structure.

Disjoint-set data structure

Elements are partitioned into *disjoint* sets.

A union-find algorithm performs two operations:

- *Find*: determine which set a certain element is in.
- *Union*: join two different sets.

Each element links to the head of its list, so that *Find* takes $O(1)$.

The head of the list carries a pointer to the size of the list.

When to sets are joined, the *larger* set L subsumes the *smaller* set S:

- the tail of L is made to point to the head of S;
- elements in S link to the head of L; and
- the size of the joint list is updated.

Kruskal's algorithm: time complexity

Let the graph contain n nodes and m edges.

Sorting the list of edges takes $O(m \cdot \log m) = O(m \cdot \log n)$.

Determining for each edge whether it is between different fragments in total takes $O(m)$.

When two fragments are joined, only nodes in the smaller fragment update their link to the head.

So each node updates its link at most $O(\log n)$ times.

Therefore joining fragments in total takes at most $O(n \cdot \log n)$.

So the overall worst-case time complexity is $O(m \cdot \log n)$.

Undecidability and NP-completeness

Some (at first sight simple) problems with an infinite possible solutions space are undecidable.

No algorithm exists to always solve such problems in finite time.

Some problems with a finite possible solutions space (such as traveling salesman and 0-1 knapsack) are NP-complete.

No efficient algorithms seem to exist to solve such problems.

First we need to dive into the foundations of computing.

Strings

A computer program takes an input string and produces an output string.

A computer program consists of a string of characters.

Some computer programs have themself as meaningful input.

For example, a C program to *parse* C programs (i.e., to check if they are syntactically correct).

Strings

A string, denoted by u, v, w, is

- a finite sequence of symbols, denoted by a, b, c,
- from a (non-empty) finite input alphabet Σ.

λ is the empty string.

Σ^* denotes the set of strings, and Σ^+ the set of non-empty strings.

Formal language

A (formal) language is a *set* of strings.

Examples:

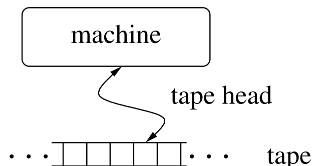
The set of pairs of input and resulting output string for a given computer program.

The set of parsable C programs.

Turing machine

A Turing machine mechanically operates on a tape.

The machine reads and writes symbols on the tape, one at a time, and steps to the left or the right.



The *input string* is in the language accepted by the Turing machine if, starting from the *initial* state, a *final* state is reached.

Turing machine

The tape is a string over a tape alphabet.

The initial input string is over a more restrictive input alphabet.

The tape serves as *unbounded* memory capacity.

The input string is surrounded at both sides by infinitely many $\Box$'s.

The tape is inspected by a tape head that repeatedly:

- reads a symbol of the tape;
- overwrites this symbol; and
- moves one place to the left or right.

Turing machine

Q is a finite collection of states.

The transition function δ has the form

$$\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$$

δ is a *partial* function: $\delta(q, a)$ needn't always be defined.

$\delta(q, a) = (r, b, L)$ means: if the machine is in state q, and the head reads a on the tape, then

- the machine goes to state r,
- the a is replaced by b, and
- the head moves one position to the left.

With $\delta(q, a) = (r, b, R)$, the head moves one position to the right.

Halt state

We abbreviate “Turing machine” to TM.

If a TM in state q reads a on the tape and $\delta(q, a)$ is undefined, then it has reached a halt state.

An execution of a TM, on a certain input string, may never reach a halt state.

Question: Suppose each final state q_f is turned into a halt state, meaning that all transitions $\delta(q_f, _)$ are discarded.

Can this have an impact on the accepted language ?

Turing machine: example

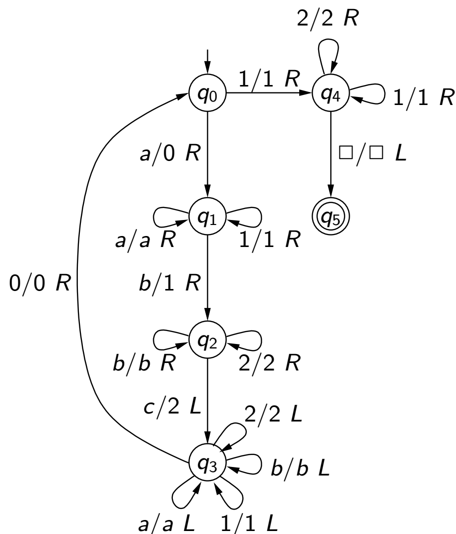
We construct a TM M with $L(M) = \{a^n b^n c^n \mid n \geq 1\}$
(i.e., each string consists of n a's followed by n b's followed by n c's).

Idea: Repeatedly replace an a by 0, a b by 1 and a c by 2.

With an input string $a^n b^n c^n$, all a's, b's and c's will vanish.

- q_0: Read a, replace it by 0, move to the right, go to q_1.
- q_1: Walk to the right until b is read, replace it by 1, move to the right, go to q_2.
- q_2: Walk to the right until c is read, replace it by 2, go to q_3.
- q_3: Walk to the left until 0 is read, move to the right, go back to q_0.
- If in q_0 1 is read, go to q_4.
- q_4: Walk to the right to check if any a's, b's or c's are left. If a $\Box$ is encountered, go to the final state q_5.

Turing machine: example



$$q_0 aabbcc$$
$$\vdash^+ \; 0q_0a1b2c$$
$$\vdash^+ \; 00q_01122$$
$$\vdash^+ \; 00112q_52$$

$$q_0 aabbbcc$$
$$\vdash^+ \; 0q_0a1bb2c$$
$$\vdash^+ \; 00q_011b22$$
$$\vdash^+ \; 0011q_4b22$$

Questions

Why do we replace a and b by different values ?

Answer: Else we don't know when to go from q_3 to q_0.

Could we replace b and c by the same value ?

Answer: No, for else e.g. *aabcbc* would be accepted.

Question

Specify a TM that only accepts sequences of a's of *odd* length.

Answer: Let $F = \{q_2\}$.

$$
\begin{aligned}
\delta(q_0, a) &= (q_1, a, R) \\
\delta(q_1, a) &= (q_0, a, R) \\
\delta(q_1, \Box) &= (q_2, \Box, L)
\end{aligned}
$$

Turing machine

A deterministic Turing machine is a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, \Box, F)$.

- Q a finite set of states
- $\Sigma \subseteq \Gamma \setminus \{\Box\}$ (i.e., Γ minus $\Box$) the input alphabet
- Γ the tape alphabet
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ the partial transition function
- q_0 the start state
- $\Box \in \Gamma$
- $F \subseteq Q$ the set of final states

Turing machine

A configuration of a TM is a string vqw ($q \in Q$, $v, w \in \Gamma^*$), where:

- the machine is in state q;
- the tape consists of vw (flanked by infinitely many $\Box$'s); and
- the head is on the first symbol of w.

$$
\begin{array}{lcllcl}
vqaw & \vdash & vbq'w & \text{if } \delta(q,a) & = & (q',b,R) \\
vq & \vdash & vbq' & \text{if } \delta(q,\Box) & = & (q',b,R) \\
 & & & & & \\
vcqaw & \vdash & vq'cbw & \text{if } \delta(q,a) & = & (q',b,L) \\
vcq & \vdash & vq'cb & \text{if } \delta(q,\Box) & = & (q',b,L)
\end{array}
$$

Turing machine

$\vdash^*$ denotes the transitive-reflexive closure of $\vdash$.

The language $L(M)$ accepted by TM $M = (Q, \Sigma, \Gamma, \delta, q_0, \Box, F)$ is

$$\{w \in \Sigma^+ \mid \exists q \in F,\ u, v \in \Gamma^* :\ q_0 w \vdash^* uqv\}$$

Note: $\lambda \notin L(M)$

Recursively enumerable languages

A language L is recursively enumerable if $L \setminus \{\lambda\}$ (i.e., L minus the empty string) is accepted by a TM.

Extensions of TMs, such as

- multiple tapes
- nondeterminism
- $\cdots$

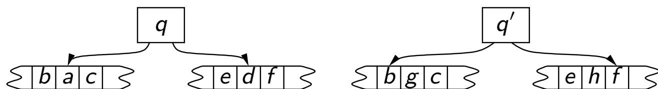
give *no* additional expressive power.

Church-Turing thesis: every computation by a computer can be simulated by a (deterministic) TM.

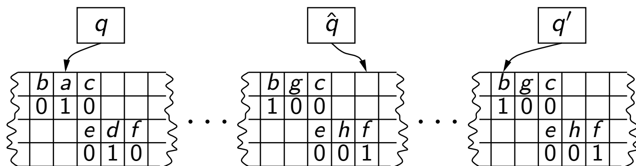
TMs with two tapes

Theorem: A TM with two tapes can be simulated by a TM with one tape.

Example: $\delta(q, a, d) = (q', g, h, L, R)$



With a single tape, this transition is simulated by 9 transitions:



The difference in time complexity between a TM with one tape and with multiple tapes is a polynomial factor.

Nondeterministic TMs

A nondeterministic TM has as transition function

$$\delta : Q \times \Gamma \to 2^{Q \times \Gamma \times \{L,R\}}$$

with $\delta(q, a)$ a *finite* subset of $Q \times \Gamma \times \{L, R\}$ for all q, a.

Nondeterministic TMs

Theorem: A nondeterministic TM N can be simulated by a deterministic TM M.

Proof: By a breadth-first search, the nondeterministic executions of N can be performed in parallel in a deterministic fashion.

M keeps this breadth-first search on its tape in the form of a queue.

The difference in time complexity between N and M is (as far as known) an exponential factor.

Universal TM

A computer can execute every (valid) program on every input.

Likewise a universal TM exists, that takes as input a TM M and an input string w, and executes M on w.

The universal TM uses three tapes, on which initially are encoded: (1) the δ-function of M, (2) the string w with the tape head on the first symbol, and (3) the start state q_0.

The universal TM repeatedly performs the following steps:

- read on the 3rd tape the state, and
  on the 2nd tape the symbol under the tape head;
- search the 1st tape for the corresponding transition (if present);
- overwrite the state on the 3rd tape;
- overwrite the symbol on the 2nd tape, and move the tape head to the left or right.

Set operations

union	$L_1 \cup L_2$	$\{x \mid x \in L_1 \lor x \in L_2\}$
intersection	$L_1 \cap L_2$	$\{x \mid x \in L_1 \land x \in L_2\}$
difference	$L_1 \setminus L_2$	$\{x \mid x \in L_1 \land x \notin L_2\}$
complement	$\overline{L}$	$\{x \mid x \notin L\}$

Recursive languages

Recursively enumerable languages are closed under $\cup$ and $\cap$.

But the complement $\overline{L}$ of a recursively enumerable language L isn't always recursively enumerable.

The proof uses a recursive enumeration $M_1, M_2, M_3, \ldots$ of all TMs:

- Each TM can be represented as an input string.
- A parsing algorithm checks for each possible input string whether it represents a TM.
- If so, add this TM to the enumeration.

Recursive languages

A language L is recursive if both L and $\overline{L}$ are recursively enumerable.

Theorem: Not every recursively enumerable language is recursive.

Proof: Let $M_1, M_2, M_3, \ldots$ be a recursive enumeration of all TMs.

We define $L = \{a^i \mid a^i \in L(M_i), i \geq 1\}$.

L is recursively enumerable: given an $i \geq 1$, look up M_i, run M_i on a^i, and accept a^i if a final state is reached.

Suppose toward a contradiction that $\overline{L}$ is recursively enumerable, i.e., $\overline{L} = L(M_k)$ for a certain $k \geq 1$.

Then $a^k \in \overline{L} \;\Leftrightarrow\; a^k \in L(M_k) \;\Leftrightarrow\; a^k \in L \;\Leftrightarrow\; a^k \notin \overline{L}$.

Contradiction, so $\overline{L}$ isn't recursively enumerable.

Diagonalization argument

The construction of $\overline{L}$ can be depicted as follows.

	M_1	M_2	M_3	M_4	$\cdots$
a^1	**X**				
a^2		**X**			
a^3			**X**		
a^4				**X**	
$\vdots$					

This is similar to the argumentation (by Georg Cantor) that the real numbers aren't countable.

Halting problem (1936)

If a TM in state q reads a on the tape, while $\delta(q, a)$ is undefined, then it has reached a *halt state*.

A final state q is supposed to be a halt state for all a.

(Removing such instructions $\delta(q, a)$ doesn't change the language accepted by the TM.)

Halting problem: Will (deterministic) TM M not reach a halt state on input string w?

The *language* corresponding to the halting problem contains string (M, w) if and only if TM M doesn't reach a halt state on input w.

Question

Suppose the halting problem is decidable, meaning that some TM can determine for each TM M and string w whether M doesn't halt on w.

Argue that then each recursively enumerable language would be recursive.

In other words, argue how we could then decide that $w \notin L(M)$.

Halting problem is undecidable

Theorem: The halting problem is undecidable.

Proof: Suppose a TM $\mathcal{H}$ exists that, given a TM and an input string, determines whether no halt state will be reached.

Given a TM M and an input string w.

Execute in parallel M on w and $\mathcal{H}$ on (M, w).

One of the following three cases is guaranteed to occur:

- M with input w reaches a final state: $w \in L(M)$.
- M with input w reaches a non-final halt state: $w \notin L(M)$.
- $\mathcal{H}$ with input (M, w) reaches a final state: $w \notin L(M)$.

So $\overline{L(M)}$ is recursively enumerable.

Contradiction, as not all recursively enumerable languages are recursive.

Halting problem is undecidable - alternative proof

Proof: Suppose a TM $\mathcal{H}$ exists that, given a TM and an input string, determines whether no halt state will be reached.

Build a TM $\mathcal{I}$ that takes as input a TM M, and asks $\mathcal{H}$ whether *M doesn't reach a halt state on input M*.

- If a halt state is reached, then $\mathcal{I}$ starts an infinite loop.
- If no halt state is reached, then $\mathcal{I}$ halts immediately.

Contradiction: $\mathcal{I}$ reaches a halt state on input $\mathcal{I}$
$\Leftrightarrow$ $\mathcal{I}$ doesn't reach a halt state on input $\mathcal{I}$.

Post correspondence problem (1946)

PCP: Given two sequences of n strings over Σ:

$$w_1, \ldots, w_n \qquad \text{and} \qquad v_1, \ldots, v_n$$

Is there a non-empty sequence of indices $j, \ldots, k$ such that

$$w_j \cdots w_k \quad = \quad v_j \cdots v_k \quad ?$$

Emil Post (1897-1954)

Question: Give a solution for

$$\begin{array}{lllllllll} w_1 & = & 01 & w_2 & = & 1 & w_3 & = & 110 \\ v_1 & = & 100 & v_2 & = & 011 & v_3 & = & 1 \end{array}$$

Post correspondence problem is undecidable

We will prove that the PCP is undecidable.

If there is a solution, it can be found by trying all (infinitely many) possible solutions.

The problem is if there isn't a solution: when do we stop searching ?

Modified Post correspondence problem

We first prove that it is undecidable whether the Modified PCP (MPCP) has no solution.

MPCP: Given two sequences of n strings over Σ:

$$w_1, \ldots, w_n \qquad \text{en} \qquad v_1, \ldots, v_n$$

Is there a non-empty sequence of indices $j, \ldots, k$ such that

$$w_1 w_j \cdots w_k \quad = \quad v_1 v_j \cdots v_k \quad ?$$

MPCP: undecidability

Theorem: If it were decidable for any instance of the MPCP whether it has no solution, then the question $w \notin L(M)$? would be decidable.

Proof: We define the following sequences of strings:

λ	$\# q_0 w$	
d	d	for any $d \in \Gamma \cup \{\#\}$
qa	br	if $\delta(q, a) = (r, b, R)$
$q\#$	$br\#$	if $\delta(q, \Box) = (r, b, R)$
eqa	reb	if $\delta(q, a) = (r, b, L)$, for any $e \in \Gamma$
$\# qa$	$\# r \Box b$	if $\delta(q, a) = (r, b, L)$
$q_f e$	q_f	for any $e \in \Gamma$
$e q_f$	q_f	for any $e \in \Gamma$
$\# q_f$	λ	

There is a MPCP solution if and only if $w \in L(M)$.

Conclusion: The MPCP is undecidable.

MPCP: example

Consider the following TM M, with $\Sigma = \{a, b, c\}$:

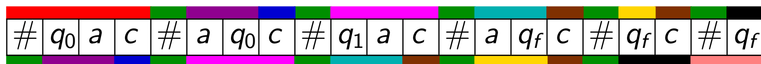
$$\delta(q_0, a) = (q_0, a, R) \qquad \delta(q_0, c) = (q_1, c, L)$$
$$\delta(q_0, b) = (q_0, b, R) \qquad \delta(q_1, a) = (q_f, a, R)$$

$ac \in L(M)$? is associated to the following instance of the MPCP:

λ	$\#q_0ac$	
d	d	for any $d \in \{a, b, c, \Box, \#\}$
q_0a	aq_0	
q_0b	bq_0	
eq_0c	q_1ec	for any $e \in \{a, b, c, \Box\}$
$\#q_0c$	$\#q_1\Box c$	
q_1a	aq_f	
q_fe	q_f	for any $e \in \{a, b, c, \Box\}$
eq_f	q_f	for any $e \in \{a, b, c, \Box\}$
$\#q_f$	λ	

MPCP: example

This instance of the MPCP has a solution, mimicking the execution of TM M on string ac toward final state q_f.



Question: Which instance of the MPCP is associated to $c \in L(M)$?

Does this instance of the MPCP have a solution ?

Reductions

$L_1 \subseteq \Sigma_1^*$ is reducible to $L_2 \subseteq \Sigma_2^*$ if there is a computable function $f : \Sigma_1^* \to \Sigma_2^*$ such that:

$$v \in L_1 \;\Leftrightarrow\; f(v) \in L_2$$

If "$w \in L_2$?" can be decided, then so can "$v \in L_1$?".

Vice versa, if "$v \in L_1$?" is undecidable, then so is "$w \in L_2$?".

We show that the MPCP is reducible to the PCP, which implies that the PCP is undecidable.

PCP: undecidability

Theorem: It is undecidable if any instance of the PCP has no solution.

Proof: Given any instance of the MPCP: $w_1, \ldots, w_n$ and $v_1, \ldots, v_n$, with $w_i = a_{i1} \cdots a_{im_i}$ and $v_i = b_{i1} \cdots b_{ir_i}$ (and $m_i + r_i > 0$) for all i.

We define two new sequences $y_0, \ldots, y_{n+1}$ and $z_0, \ldots, z_{n+1}$:

$$y_0 = \$y_1 \qquad y_i = a_{i1}\$a_{i2}\$\cdots a_{im_i}\$ \quad (1 \le i \le n) \qquad y_{n+1} = \#$$
$$z_0 = z_1 \qquad z_i = \$b_{i1}\$b_{i2}\cdots\$b_{ir_i} \quad (1 \le i \le n) \qquad z_{n+1} = \$\#$$

\$ and # are fresh, so each PCP solution is of the form

$$y_0 y_j \cdots y_k y_{n+1} = z_0 z_j \cdots z_k z_{n+1}$$

This is a PCP solution if and only if

$$w_1 w_j \cdots w_k = v_1 v_j \cdots v_k$$

is a MPCP solution for the w_i's and v_i's.

Since the MPCP is undecidable, the PCP is undecidable too.

Reduction of MPCP tot PCP: example

Consider the following instance of the MPCP:

$$\begin{aligned} w_1 &= 11 & w_2 &= 1 \\ v_1 &= 1 & v_2 &= 11 \end{aligned}$$

To which instance of the PCP does it reduce ?

$$\begin{aligned} y_0 &= \$1\$1\$ & y_1 &= 1\$1\$ & y_2 &= 1\$ & y_3 &= \# \\ z_0 &= \$1 & z_1 &= \$1 & z_2 &= \$1\$1 & z_3 &= \$\# \end{aligned}$$

The original MPCP instance has a solution if and only if the resulting PCP instance has a solution.

Semi-decidability

The halting problem and the PCP are semi-decidable.

Namely, these problems are decidable if:

- the TM reaches a halt state on the given input, or
- the PCP instance has a solution.

More undecidable problems

In 1900, David Hilbert (1862-1941) formulated 23 mathematical problems.

Diophantine equations consist of polynomials with one or more variables, and coefficients in $\mathbb{Z}$. For example:

$$\begin{aligned} 3X^2Y - 7Y^2Z^3 - 18 &= 0 \\ -7Y^2 + 8Z^3 &= 0 \end{aligned}$$

Hilbert's 10th problem: Give an algorithm to determine whether a system of Diophantine equations has a solution in $\mathbb{Z}$.

In 1970, Yuri Matiyasevich proved that this problem is undecidable.

Complexity classes P and NP

A nondeterministic TM M is polynomial-time-bounded
if there is a polynomial $p(k)$ such that for each input w,
M always reaches a halt state in at most $p(|w|)$ steps.

- $L(M) \in$ NP
- if M is deterministic, then $L(M) \in$ P

P $\subseteq$ NP, but unknown is whether P $=$ NP.

Recall that simulating a nondeterministic TM by a deterministic TM takes an exponential amount of time.

Problems in NP

The language that belongs to a decision problem, consists of strings that represent an instance of the problem, and have the outcome *yes*.

Examples:

The question whether the traveling salesman problem has a solution $\leq k$ (for a certain k) is in NP.

The question whether a number is not prime is in NP.

Surprisingly, this question turned out to be in P.

(Agrawal, Kayal, Saxena, 2002)

Problems in NP

Intuitively, a decision problem is in NP if:

- every instance has finitely many possible solutions, and
- it can be checked in polynomial time whether or not a possible solution for an instance is correct.

Question: Given such a decision problem.

Build a polynomial-time-bounded nondeterministic TM that accepts exactly all solvable problem instances.

The question if a number isn't prime is in NP

Proof: We build a *polynomial-time-bounded* TM that accepts exactly all non-primes n.

- Choose *nondeterministically* a number $k \in \{2, 3, \ldots, \lfloor \frac{n}{2} \rfloor\}$.
- Check whether k divides n. If so, accept.

NP-completeness

$L_1 \subseteq \Sigma_1^*$ is reducible to $L_2 \subseteq \Sigma_2^*$ if there is a computable function $f : \Sigma_1^* \to \Sigma_2^*$ such that:

$$v \in L_1 \;\Leftrightarrow\; f(v) \in L_2$$

If "$w \in L_2$?" can be decided, then so can "$v \in L_1$?".

L_1 is polynomial-time reducible to L_2 if the function $f : \Sigma_1^* \to \Sigma_2^*$ can be computed in polynomial time.

If "$w \in L_2$?" can be decided in polynomial time, then so can "$v \in L_1$?".

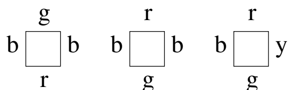
A language $L \in$ NP is NP-complete if *each language in NP* is polynomial-time reducible to L.

Bounded tiling problem

Given a finite set of types of 1×1 tiles, with a "color" at each side.

There are infinitely many tiles of each type.

Example:

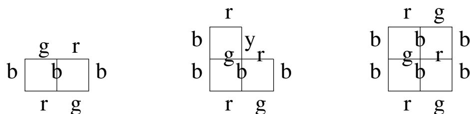


When tiling a surface, touching sides of neighboring tiles must have the same color. (Tile types can't be rotated.)

Bounded tiling problem: Given a value n and a first row of n tiles.

Can the $n \times n$ surface be tiled ?

Example: $n = 2$.



Bounded tiling problem is NP-complete

Theorem: The bounded tiling problem is NP-complete.

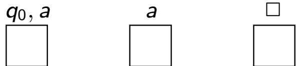
Proof: The bounded tiling problem is in NP.

- For each instance there are finitely many possible solutions.
- It can be checked in polynomial time whether a possible solution is correct.

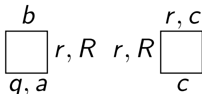
Let the nondeterministic TM M be polynomial-time-bounded by a polynomial $p(k)$. (We assume that $p(k) \geq k$.)

We give a polynomial-time reduction from $w \in L(M)$? to the bounded tiling problem.

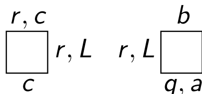
To build the first row (for all $a \in \Sigma$):



Instructions of M (for all $c \in \Gamma$):

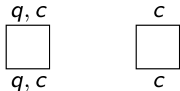


$(r, b, R) \in \delta(q, a)$

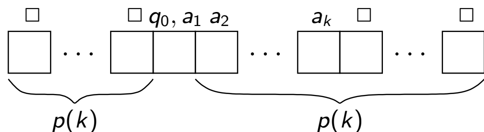


$(r, b, L) \in \delta(q, a)$

Leave the tape unchanged (for all $q \in F$ and $c \in \Gamma$):



For an input string $w = a_1 \cdots a_k$ we choose $n = 2p(k) + 1$, and as first row of tiles:



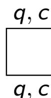
The $n \times n$ surface can be tiled $\Leftrightarrow$ $w \in L(M)$.

Namely, each tiling simulates an execution of M on input w.

Such an execution consists of at most $p(k)$ steps.

This produces a tiling of height $\leq p(k) + 1$.

Since tiles



exist only for $q \in F$, the tiling can only be completed to size $n \times n$ if the corresponding execution of M on input w reaches a final state.

Bounded tiling problem: example

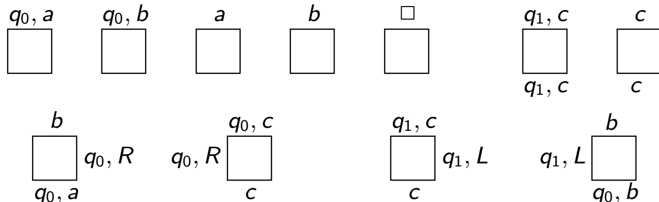
Consider TM M with $\Sigma = \{a, b\}$, $\Gamma = \Sigma \cup \{\Box\}$, $F = \{q_1\}$ and

$$\delta(q_0, a) = (q_0, b, R) \qquad \delta(q_0, b) = (q_1, b, L)$$

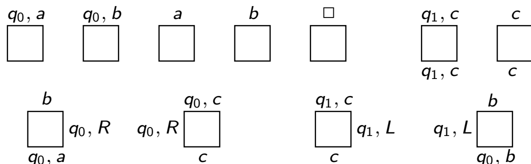
$L(M)$ consists of all strings containing a b.

At input w, M takes at most $|w|$ steps. So we take $p(k) = k$.

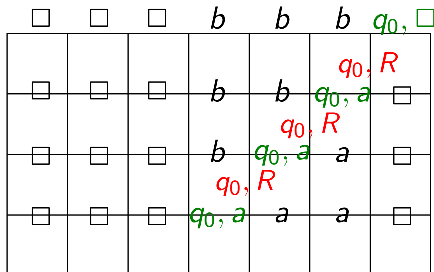
Tile types are (for all $c \in \Gamma$):



Bounded tiling problem: example

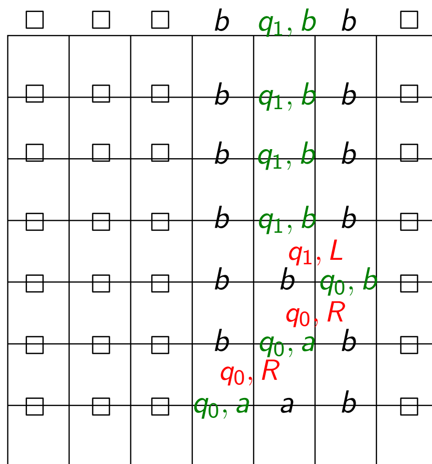


Consider the input string $aaa \notin L(M)$. Then $n = 7$.



Bounded tiling problem: example

Consider the input string $aab \in L(M)$. Then $n = 7$.



Millennium Prize Problem

Question: Suppose a polynomial-time algorithm is discovered for the bounded tiling problem.

How does this induce a polynomial-time algorithm for any problem in NP ?

So far no polynomial-time algorithm for the bounded tiling problem has been found.

It remains an open question whether such an algorithm exists.

P = NP ? is one of seven Millennium Prize Problems posed by the Clay Mathematics Institute, with a reward of one million dollar.

Polynomial-time reductions are compositional

$L_1 \subseteq \Sigma_1^*$ is polynomial-time reducible to $L_2 \subseteq \Sigma_2^*$ if there is an in polynomial time computable function $f : \Sigma_1^* \to \Sigma_2^*$ such that:

$$w \in L_1 \;\Leftrightarrow\; f(w) \in L_2$$

The composition $g \circ f : \Sigma_1^* \to \Sigma_3^*$ of polynomial-time reductions $f : \Sigma_1^* \to \Sigma_2^*$ and $g : \Sigma_2^* \to \Sigma_3^*$ is a polynomial-time reduction.

So if an NP-complete L_1 is polynomial-time reducible to $L_2 \in \mathrm{NP}$, then L_2 is NP-complete.

NP-complete problems: examples

The question whether the traveling salesman problem has a solution $\leq k$ (for a certain k) is NP-complete.

A Boolean formula consists of `true`, `false`, conjunction $\wedge$, disjunction $\vee$, negation $\neg$, and variables.

A Boolean formula Φ is satisfiable if there is a substitution of values `true` and `false` for the variables in Φ such that Φ evaluates to `true`.

The satisfiability problem of Boolean formulas is NP-complete.

Question

Is the following Boolean formula satisfiable ?

$$((x \land y) \lor (\neg x \land \neg y)) \land ((x \land \neg y) \lor (\neg x \land y))$$

How many different substitutions of truth values exist for a Boolean formula with n different variables ?

Satisfiability problem is in NP

Theorem: The question whether a Boolean formula is satisfiable is in NP.

Proof: We build a *polynomial-time-bounded* TM that accepts exactly all satisfiable Boolean formulas:

- Verify whether the input is a well-defined Boolean formula.
- Choose *nondeterministically* one of the possible substitutions of truth values for the variables in the formula.
- Perform this substitution on the formula.
- Check whether the resulting formula is *true*. If so, accept.

Satisfiability problem is NP-complete

Cook's Theorem: The satisfiability problem is NP-complete.

Proof: There exists a polynomial-time reduction from the bounded tiling problem to the satisfiability problem.

Given a finite set T of tile types, n, and a first row of tiles $t_1 \cdots t_n$.

Boolean variable $x_{k\ell t} = \texttt{true}$ (for $1 \leq k, \ell \leq n$ and $t \in T$) represents that position (k, ℓ) in the $n \times n$ tiling is occupied by a tile of type t.

There is an $n \times n$ tiling with first row of tiles $t_1 \cdots t_n$ if and only if the conjunction of the following four Boolean formulas is satisfiable.

The first row consists of $t_1 \cdots t_n$:

$$\bigwedge_{k=1}^{n} x_{k1t_k}$$

Each position holds at most one tile type:

$$\bigwedge_{k=1}^{n} \bigwedge_{\ell=1}^{n} \bigwedge_{t \neq t'} \neg(x_{k\ell t} \wedge x_{k\ell t'})$$

Horizontally touching sides have the same color:

$$\bigwedge_{k=1}^{n-1} \bigwedge_{\ell=1}^{n} \bigvee_{tt' \text{ legal}} (x_{k\ell t} \wedge x_{(k+1)\ell t'})$$

Vertically touching sides have the same color:

$$\bigwedge_{k=1}^{n} \bigwedge_{\ell=1}^{n-1} \bigvee_{\substack{t' \\ t} \text{ legal}} (x_{k\ell t} \wedge x_{k(\ell+1)t'})$$

Satisfiability problem is NP-complete

The size of the resulting Boolean formula is polynomial in n.

Hence we have constructed a polynomial-time reduction from the bounded tiling problem to the satisfiability problem.

We conclude that the satisfiability problem is NP-complete.

Question

Reduce the following instance of the bounded tiling problem to the satisfiability problem.

Tile types:

g
b b
r

r
b b
g

r
b y
g

First row:

g r
b b b
r g

Other NP-complete problems

By means of polynomial-time reductions, a wide range of problems have been shown to be NP-complete.

For example:

- 0-1 knapsack problem
- traveling salesman problem
- Hamiltonian cycle problem
- integer programming

Branch and bound

Branch-and-bound algorithms are commonly used to solve NP-complete optimization problems.

Consider a problem to maximize the value of some function $f(x)$.

(E.g., the 0-1 knapsack or traveling salesman problem.)

We build a *binary tree*, in which each node contains a subset of possible solutions.

The *root* contains all possible (correct and incorrect) solutions.

(E.g., each possible subset of items in the knapsack, or each possible subset of roads between cities.)

Branch and bound

branch: Split the set of possible solutions in a node in two parts, which are stored in the two nodes below.

Pick one item, and consider solutions that include this item separately from solutions that exclude this item.

(E.g., 1 item in/outside the knapsack, or 1 road in/outside the path.)

bound: Determine an (easily computable) upper bound of $f(x)$, for the possible solutions in a node.

prune: If a solution $f(x) = a$ is found, all intermediate nodes with an upper bound $\leq a$ are pruned away.

best-first search: branch on a node with the highest upper bound.

Branch-and-bound algorithm for the 0-1 knapsack problem

Order items on the basis of $\frac{v}{w}$ (with v their value and w their weight).

The root node contains all possible subsets of items.

That is, for each item it is undetermined whether it is selected.

branch: Pick the highest undetermined item i, and split solutions that include i from solutions that exclude i from the knapsack.

bound: Determine an upper bound for each of the two subsets, by (illegally) assuming a *fraction* of an item can be in the knapsack.

Branch-and-bound algorithm for the 0-1 knapsack problem

Example: $W = 100$

item	v	w	$\frac{v}{w}$
1	60	30	2
2	60	50	$\frac{6}{5}$
3	40	40	1
4	10	10	1
5	20	40	$\frac{1}{2}$
6	10	30	$\frac{1}{3}$
7	3	10	$\frac{3}{10}$

In each node we keep track of:

- The added value and weight of all items that were determined to be in the knapsack.
- An upper bound on the possible solutions corresponding to this node.

For instance, in the root node the overall value and weight are 0 and the upper bound is $v_1 + v_2 + \frac{1}{2} \cdot v_3 = 140$.

Branch-and-bound algorithm for the 0-1 knapsack problem

$W = 100$

item	v	w	$\frac{v}{w}$
1	60	30	2
2	60	50	$\frac{6}{5}$
3	40	40	1
4	10	10	1
5	20	40	$\frac{1}{2}$
6	10	30	$\frac{1}{3}$
7	3	10	$\frac{3}{10}$

$$v_1 + v_2 + \tfrac{1}{2} \cdot v_3 = 140$$
$$v_2 + v_3 + v_4 = 110$$
$$v_1 + v_3 + v_4 + \tfrac{1}{2} \cdot v_5 = 120$$
$$v_1 + v_2 + v_4 + \tfrac{1}{4} \cdot v_5 = 135$$
$$v_1 + v_2 + \tfrac{1}{2} \cdot v_5 = 130$$
$$v_1 + v_2 + v_4 + \tfrac{1}{3} \cdot v_6 = 133\tfrac{1}{3}$$
$$v_1 + v_2 + v_4 + v_7 = 133$$

Branch-and-bound algorithm for the 0-1 knapsack problem

No more nodes can be added below the bottom node, because at this point the weight 100 has been exhausted.

Since all "pending" nodes have an upper bound smaller than 133, they are pruned away, and the algorithm terminates with as outcome:

- v_1 v_2 v_4 v_7 in the knapsack
- total value 133

The best-first search approach of branch and bound often takes linear time, but in the worst case takes exponential time.

Branch-and-bound algorithm for the 0-1 knapsack problem

Let $W = 2{\cdot}n - 1$.

There are n items of value $2 + \frac{1}{n}$ and weight 2.

There is 1 item of value $2{\cdot}n - 1$ and weight $2{\cdot}n - 1$.

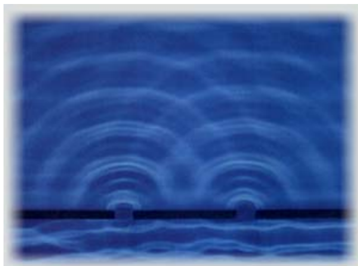
The n small items are ordered before the large item, because $\frac{2+\frac{1}{n}}{2} > 1$.

Therefore the branch-and-bound algorithm will first exhaustively try to fit a subset of first n items into the knapsack, every time yielding the total value $(n-1){\cdot}2 = 2{\cdot}n - 2$.

Only at the very end the optimal solution is found:
Put only the last element in the knapsack, with total value $2{\cdot}n - 1$.

Huygens' principle

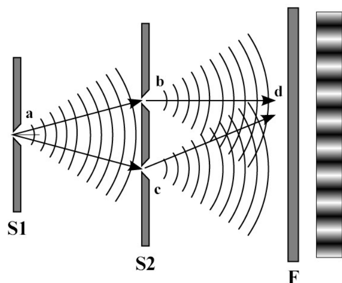
Each point on a wave front becomes a source of a spherical wave.



Double-slit experiment of Young

Christiaan Huygens predicted in 1678 that light behaves as a wave.

Thomas Young showed in 1805 that this is indeed the case.



But if light is measured at the slits, its particles (photons) travel in a straight line.

Quantum mechanics

An elementary particle can behave as a wave or as a particle.

Superposition: A particle can simultaneously be in a range of states, with some probability distribution.

Interaction with an observer causes a particle to assume a single state.

Superposition is represented using *complex* numbers.

Complex numbers

"God made the natural numbers; all else is the work of man"

(Leopold Kronecker, 1886)

$$\begin{array}{rclcrclc} x+1 & = & 0 & \quad & x & = & -1 & \quad \mathbb{Z} \\ 2x & = & 1 & & x & = & \frac{1}{2} & \quad \mathbb{Q} \\ x^2 & = & 2 & & x & = & \sqrt{2} & \quad \mathbb{R} \\ x^2 & = & -1 & & x & = & i & \quad \mathbb{C} \end{array}$$

A complex number in $\mathbb{C}$ is of the form $a + bi$ with $a, b \in \mathbb{R}$.

Fundamental theorem of algebra: $\mathbb{C}$ is algebraically closed !

Qubits

A qubit (short for *quantum bit*) is in a superposition

$$\alpha_0 |0\rangle + \alpha_1 |1\rangle$$

with $\alpha_0, \alpha_1 \in \mathbb{C}$, where $|\alpha_0|^2 + |\alpha_1|^2 = 1$.

At interaction with an observer, the qubit takes on the value 0 with probability $|\alpha_0|^2$, and the value 1 with probability $|\alpha_1|^2$.

After such an interaction the qubit is no longer in superposition, but in a single state 0 or 1.

For simplicity we assume that $\alpha_0, \alpha_1 \in \mathbb{R}$. (They can be negative !)

2-qubits

A system of two qubits has four states:

$$\alpha_{00}|00\rangle + \alpha_{01}|01\rangle + \alpha_{10}|10\rangle + \alpha_{11}|11\rangle$$

where $\alpha_{00}^2 + \alpha_{01}^2 + \alpha_{10}^2 + \alpha_{11}^2 = 1$.

Two *independent* qubits $\alpha_0|0\rangle + \alpha_1|1\rangle$ and $\beta_0|0\rangle + \beta_1|1\rangle$ can be described by:

$$\alpha_0{\cdot}\beta_0|00\rangle + \alpha_0{\cdot}\beta_1|01\rangle + \alpha_1{\cdot}\beta_0|10\rangle + \alpha_1{\cdot}\beta_1|11\rangle$$

Entanglement of qubits

Example: Consider the 2-qubit $\frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|11\rangle$.

When one of the qubits is observed, *both* qubits assume the same state.

Such a relation between superpositions is called entanglement.

Entanglement can occur at the decay of an elementary particle into multiple particles and is preserved when these particles are no longer close to each other.

John Bell confirmed this phenomenon experimentally in 1964.

EPR paradox (1935)

Einstein, Podolsky and Rosen formulated the following paradox.

Let two entangled particles travel to different corners of the universe.

How can the superpositions of these particles be instantly related ?

According to relativity theory, nothing travels faster than light.

> "*If someone tells you they understand quantum mechanics, then all you've learned is that you've met a liar.*"
>
> (Richard Feynman)

Measuring one qubit of a 2-qubit

Consider a 2-qubit

$$\alpha_{00}\,|00\rangle\ +\ \alpha_{01}\,|01\rangle\ +\ \alpha_{10}\,|10\rangle\ +\ \alpha_{11}\,|11\rangle$$

with $\alpha_{00}^2 + \alpha_{01}^2 + \alpha_{10}^2 + \alpha_{11}^2 = 1$.

If for instance the first of these qubits is measured with outcome 0, then the resulting superposition of the second qubit is

$$\tfrac{1}{\sqrt{\alpha_{00}^2+\alpha_{01}^2}}(\alpha_{00}\,|0\rangle\ +\ \alpha_{01}\,|1\rangle)$$

Question: Suppose the second qubit is measured with outcome 1.

What is then the resulting superposition of the first qubit ?

Questions

Question: What does a 2×2 matrix represent? For example,

$$\begin{pmatrix} 1 & 3 \\ -2 & 1 \end{pmatrix}$$

Answer: A *linear* mapping from $\mathbb{R}^2$ to $\mathbb{R}^2$.

Question: What do the columns of this matrix express?

Answer: The images of the two base vectors.

The example matrix maps $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$ to $\begin{pmatrix} 1 \\ -2 \end{pmatrix}$, and $\begin{pmatrix} 0 \\ 1 \end{pmatrix}$ to $\begin{pmatrix} 3 \\ 1 \end{pmatrix}$.

Question

What is the image of $\begin{pmatrix} -4 \\ 1 \end{pmatrix}$ under the matrix on the previous slide ?

Matrices

The 2×2 matrix

$$\begin{pmatrix} \beta_{00} & \beta_{10} \\ \beta_{01} & \beta_{11} \end{pmatrix}$$

maps base vector $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$ to $\begin{pmatrix} \beta_{00} \\ \beta_{01} \end{pmatrix}$, and base vector $\begin{pmatrix} 0 \\ 1 \end{pmatrix}$ to $\begin{pmatrix} \beta_{10} \\ \beta_{11} \end{pmatrix}$.

A matrix is a *linear* mapping:

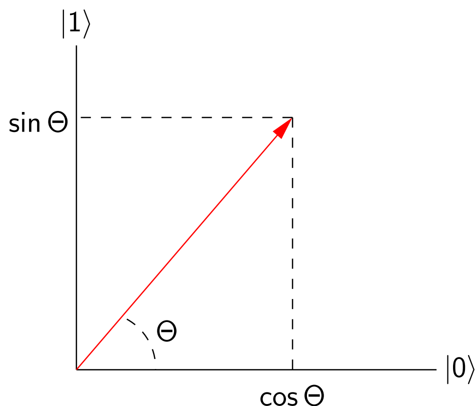
vector $\begin{pmatrix} \alpha_0 \\ \alpha_1 \end{pmatrix}$ is by the matrix mapped to $\alpha_0 \begin{pmatrix} \beta_{00} \\ \beta_{01} \end{pmatrix} + \alpha_1 \begin{pmatrix} \beta_{10} \\ \beta_{11} \end{pmatrix}$.

In other words,

$$\begin{pmatrix} \beta_{00} & \beta_{10} \\ \beta_{01} & \beta_{11} \end{pmatrix} \begin{pmatrix} \alpha_0 \\ \alpha_1 \end{pmatrix} = \begin{pmatrix} \alpha_0 \cdot \beta_{00} + \alpha_1 \cdot \beta_{10} \\ \alpha_0 \cdot \beta_{01} + \alpha_1 \cdot \beta_{11} \end{pmatrix}$$

Qubit as a vector

A qubit can be interpreted as a vector of length 1:



$\sin^2\Theta + \cos^2\Theta = 1$

Quantum operations

Physically, on a qubit one can perform a *quantum operation*

$$\begin{pmatrix} \beta_{00} & \beta_{10} \\ \beta_{01} & \beta_{11} \end{pmatrix}$$

which maps $|0\rangle$ to $\beta_{00}|0\rangle + \beta_{01}|1\rangle$ and $|1\rangle$ to $\beta_{10}|0\rangle + \beta_{11}|1\rangle$.

This operation maps a qubit $\alpha_0\,|0\rangle \;+\; \alpha_1\,|1\rangle$ to

$$(\alpha_0{\cdot}\beta_{00} + \alpha_1{\cdot}\beta_{10})\,|0\rangle \;+\; (\alpha_0{\cdot}\beta_{01} + \alpha_1{\cdot}\beta_{11})\,|1\rangle$$

Quantum operations are invertible and preserve probability mass 1.

Unitary matrices

A matrix is unitary if its columns are orthonormal:

they have *length 1* and are *orthogonal* to each other.

$\begin{pmatrix} \beta_{00} & \beta_{10} \\ \beta_{01} & \beta_{11} \end{pmatrix}$ is unitary if and only if:

$$\begin{aligned} \beta_{00}^2 + \beta_{01}^2 &= 1 \\ \beta_{10}^2 + \beta_{11}^2 &= 1 \\ \beta_{00}\cdot\beta_{10} + \beta_{01}\cdot\beta_{11} &= 0 \end{aligned}$$

Unitary matrices are *invertible* and *preserve length and angle size*.

Typical examples are *rotations* and *reflections*.

The composition of unitary matrices is again a unitary matrix.

Quantum operations: example

A quantum operation can be applied to a single qubit of an entangled pair of qubits.

Example: Consider the entangled 2-qubit $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$.

Apply to the first qubit a rotation of $\frac{\pi}{8}$ (22.5°):

$$\begin{pmatrix} \cos\frac{\pi}{8} & -\sin\frac{\pi}{8} \\ \sin\frac{\pi}{8} & \cos\frac{\pi}{8} \end{pmatrix}$$

The superposition of the 2-qubit then becomes

$$\frac{1}{\sqrt{2}}(\cos\tfrac{\pi}{8}|00\rangle - \sin\tfrac{\pi}{8}|01\rangle + \sin\tfrac{\pi}{8}|10\rangle + \cos\tfrac{\pi}{8}|11\rangle)$$

Quantum operations: example

$$\frac{1}{\sqrt{2}}\left(\cos\frac{\pi}{8}\,|00\rangle\ -\ \sin\frac{\pi}{8}\,|01\rangle\ +\ \sin\frac{\pi}{8}\,|10\rangle\ +\ \cos\frac{\pi}{8}\,|11\rangle\right)$$

Apply to the second qubit a rotation of $-\frac{\pi}{8}$:

$$\begin{pmatrix} \cos\frac{\pi}{8} & \sin\frac{\pi}{8} \\ -\sin\frac{\pi}{8} & \cos\frac{\pi}{8} \end{pmatrix}$$

The resulting superposition of the 2-qubit is
(independent from the order in which the rotations are applied):

$$\frac{1}{\sqrt{2}}\left(\left(\cos^2\frac{\pi}{8}-\sin^2\frac{\pi}{8}\right)|00\rangle\ -\ 2\cdot\sin\frac{\pi}{8}\cdot\cos\frac{\pi}{8}\,|01\rangle\right.$$
$$\left.+\ 2\cdot\sin\frac{\pi}{8}\cdot\cos\frac{\pi}{8}\,|10\rangle\ +\ \left(\cos^2\frac{\pi}{8}-\sin^2\frac{\pi}{8}\right)|11\rangle\right)$$

Parity game

Alice and Bob each *get* a randomly chosen bit, x and y.

They *answer*, independent of each other, with a bit, a and b.

Alice and Bob win if

$$a \oplus b \;=\; x \wedge y$$

with $\oplus$ the XOR (i.e., addition modulo 2) and $\wedge$ conjunction.

$$
\begin{aligned}
0 \oplus 0 &= 0 & \qquad 0 \wedge 0 &= 0 \\
0 \oplus 1 &= 1 & \qquad 0 \wedge 1 &= 0 \\
1 \oplus 0 &= 1 & \qquad 1 \wedge 0 &= 0 \\
1 \oplus 1 &= 0 & \qquad 1 \wedge 1 &= 1
\end{aligned}
$$

On a classical computer, an *optimal* strategy is that Alice and Bob both always answer 0. They then win with probability 0.75.

Parity game: quantum solution

Alice and Bob each hold a qubit of the entangled 2-qubit

$$\frac{1}{\sqrt{2}}\left(|00\rangle + |11\rangle\right).$$

If Alice receives $x = 1$, she rotates *her* qubit with $\frac{\pi}{8}$.

If Bob receives $y = 1$, he rotates *his* qubit with $-\frac{\pi}{8}$.

Finally Alice and Bob each measure their own qubit,
and return the result as answer a and b, respectively.

With this strategy, Alice and Bob win with a probability > 0.8 !!

Parity game: quantum solution

$x = y = 0$: The superposition is $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$.

So $a \oplus b = 0$ with probability 1.

$x = 1$, $y = 0$: The superposition is

$$\frac{1}{\sqrt{2}}(\cos\frac{\pi}{8}|00\rangle - \sin\frac{\pi}{8}|01\rangle + \sin\frac{\pi}{8}|10\rangle + \cos\frac{\pi}{8}|11\rangle).$$

So $a \oplus b = 0$ with probability $\cos^2\frac{\pi}{8} > 0.85$.

$x = 0$, $y = 1$: Likewise $a \oplus b = 0$ with probability $\cos^2\frac{\pi}{8} > 0.85$.

Parity game: quantum solution

$x = y = 1$: The superposition is

$$\frac{1}{\sqrt{2}}\left(\left(\cos^2\frac{\pi}{8} - \sin^2\frac{\pi}{8}\right)|00\rangle \,-\, 2\cdot\sin\frac{\pi}{8}\cdot\cos\frac{\pi}{8}\,|01\rangle\right.$$

$$\left. +\, 2\cdot\sin\frac{\pi}{8}\cdot\cos\frac{\pi}{8}\,|10\rangle \,+\, \left(\cos^2\frac{\pi}{8} - \sin^2\frac{\pi}{8}\right)|11\rangle\right)$$

So $a \oplus b = 1$ with probability $(2\cdot\sin\frac{\pi}{8}\cdot\cos\frac{\pi}{8})^2 = \sin^2\frac{\pi}{4} = 0.5$.

Conclusion: On average, Alice and Bob win with probability > 0.8.

Simon's algorithm (1994)

Given an $f : \{0,1\}^n \to \{0,1\}^n$, such that for some $s \in \{0,1\}^n \setminus \{0^n\}$:

$$\forall x, y \in \{0,1\}^n \,(x \neq y) : \; f(x) = f(y) \;\Leftrightarrow\; x \oplus y = s$$

with $\oplus$ the bitwise XOR, i.e., addition modulo 2.

Example: $n = 3$ and $s = 011$.

$f : \{0,1\}^3 \to \{0,1\}^3$ is defined as follows:

$$\begin{array}{lllll} f(000) & = & f(011) & = & 010 \\ f(001) & = & f(010) & = & 101 \end{array} \qquad \begin{array}{lllll} f(100) & = & f(111) & = & 110 \\ f(101) & = & f(110) & = & 001 \end{array}$$

Simon's algorithm

We are only given a black box (i.e., a classical circuit) that for any input x produces $f(x)$, in polynomial time (in n).

Problem: Determine s.

A *brute force* algorithm:

Search for a pair $x \neq y$ with $f(x) = f(y)$, and compute $x \oplus y$.

On average, this takes an *exponential* amount of time (in n).

Simon's algorithm *on average* determines s in *polynomial* time (in n).

Quantum gates: NOT

Since quantum operations are unitary, they are invertable and *quantum gates* always have an equal number of inputs and outputs.

NOT: $\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$

This gate carries $\alpha_0 |0\rangle + \alpha_1 |1\rangle$ over to $\alpha_1 |0\rangle + \alpha_0 |1\rangle$.

Quantum operation on two qubits

Quantum-operations can be applied to two qubits simultaneously.

Then the quantum operation is a *unitary* 4×4-matrix:

each string in $\{0,1\} \times \{0,1\}$ is a base vector.

A unitary matrix

$$\begin{pmatrix} \beta_{00} & \beta_{10} & \beta_{20} & \beta_{30} \\ \beta_{01} & \beta_{11} & \beta_{21} & \beta_{31} \\ \beta_{02} & \beta_{12} & \beta_{22} & \beta_{32} \\ \beta_{03} & \beta_{13} & \beta_{23} & \beta_{33} \end{pmatrix}$$

gives rise to the following transformations:

$$\begin{aligned} |00\rangle &\mapsto \beta_{00}|00\rangle + \beta_{01}|01\rangle + \beta_{02}|10\rangle + \beta_{03}|11\rangle \\ |01\rangle &\mapsto \beta_{10}|00\rangle + \beta_{11}|01\rangle + \beta_{12}|10\rangle + \beta_{13}|11\rangle \\ |10\rangle &\mapsto \beta_{20}|00\rangle + \beta_{21}|01\rangle + \beta_{22}|10\rangle + \beta_{23}|11\rangle \\ |11\rangle &\mapsto \beta_{30}|00\rangle + \beta_{31}|01\rangle + \beta_{32}|10\rangle + \beta_{33}|11\rangle \end{aligned}$$

Quantum gates: Controlled NOT

A quantum operation cannot *copy* a qubit, since this requires two inputs and one output.

CNOT: $\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$

$$\begin{array}{lcl} |00\rangle & \mapsto & |00\rangle \\ |01\rangle & \mapsto & |01\rangle \\ |10\rangle & \mapsto & |11\rangle \\ |11\rangle & \mapsto & |10\rangle \end{array}$$

CNOT can be represented by $|xy\rangle \mapsto |x(x \oplus y)\rangle$.

($\oplus$ denotes the XOR, i.e., addition modulo 2.)

With CNOT, x can be copied, if we make sure y initially is $|0\rangle$.

y is some kind of working memory.

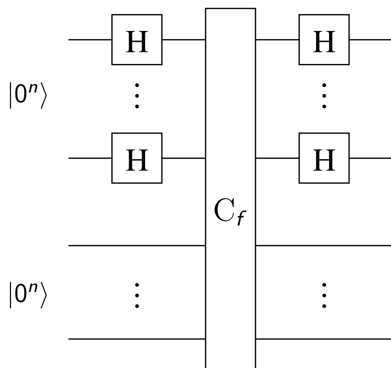
Quantum gates: Hadamard

Hadamard: $\frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$

This transformation transposes $|0\rangle$ as well as $|1\rangle$ into a superposition in which the outcomes 0 and 1 are equally likely.

Simon's algorithm

C_f transforms $[xy\rangle$ into $[x(f(x) \oplus y)\rangle$, for each $x, y \in \{0,1\}^n$.



(The circuit to compute $f(x)$ is omitted.)

Simon's algorithm

We start with $[0^n 0^n\rangle$. (The last n bits are "working memory".)

After Hadamards on the first n bits, the superposition is

$$\frac{1}{\sqrt{2^n}} \sum_{x \in \{0,1\}^n} |x\, 0^n\rangle$$

Perform C_f on the $2n$ bits: $\frac{1}{\sqrt{2^n}} \sum_{x \in \{0,1\}^n} |x\, f(x)\rangle$

Measure the last n bits: $f(x_0)$ for a certain $x_0 \in \{0,1\}^n$.

Since $f(x_0) = f(x_0 \oplus s)$, the superposition of the first n bits is

$$\frac{1}{\sqrt{2}} (|x_0\rangle + |x_0 \oplus s\rangle)$$

Simon's algorithm: example

$n = 3$, and $s = 011$, and $f : \{0,1\}^3 \to \{0,1\}^3$ is defined by:

$$\begin{array}{lllll} f(000) & = & f(011) & = & 010 \\ f(001) & = & f(010) & = & 101 \end{array} \qquad \begin{array}{lllll} f(100) & = & f(111) & = & 110 \\ f(101) & = & f(110) & = & 001 \end{array}$$

We start with $|000000\rangle$, and perform Hadamards on the first three bits.

$$\frac{1}{\sqrt{8}}(|000000\rangle + |001000\rangle + |010000\rangle + |011000\rangle + |100000\rangle + |101000\rangle + |110000\rangle + |111000\rangle)$$

Question: What is the resulting superposition if we now apply C_f ?

Simon's algorithm: example

Performing C_f on the six bits yields:

$$\frac{1}{\sqrt{8}}(|000010\rangle + |001101\rangle + |010101\rangle + |011010\rangle$$
$$+ |100110\rangle + |101001\rangle + |110001\rangle + |111110\rangle)$$

Read the last three bits. There are four possible answers:

010 101 110 001

Suppose we read 110. The superposition of the first three bits then is

$$\frac{1}{\sqrt{2}}(|100\rangle + |111\rangle)$$

Question: What is the resulting superposition if we read 101 ?

Simon's algorithm

The probability that a Hadamard $\frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$ transforms a $b \in \{0,1\}$ into a $b' \in \{0,1\}$ is $\frac{-1}{\sqrt{2}}$ if $b = b' = 1$, and $\frac{1}{\sqrt{2}}$ otherwise.

n Hadamards on a string $x \in \{0,1\}^n$ produce each possible string of length n with equal probability.

A string z is produced with a $+$ or $-$ depending on how many 1's the strings x and z have in common.

If x and z have an *even* number of 1's in common, z is produced.

If x and z have an *odd* number of 1's in common, $-z$ is produced.

Simon's algorithm

If s and z have an *even* number of 1's in common, then Hadamards on x_0 and $x_0 \oplus s$ produce z with the *same* sign.

If s and z have an *odd* number of 1's in common, then Hadamards on x_0 and $x_0 \oplus s$ produce z with the *opposite* sign.

Example: $s = 011$. Suppose $x_0 = 100$, and so $x_0 \oplus s = 111$.

- Three Hadamards on x_0 produce -101.

  Three Hadamards on $x_0 \oplus s$ produce 101.

- Three Hadamards on x_0 produce -111.

  Three Hadamards on $x_0 \oplus s$ also produce -111.

Hence, the n Hadamards on $\frac{1}{\sqrt{2}}(|x_0\rangle + |x_0 \oplus s\rangle)$ produce those z with $\langle s, z\rangle = 0 \bmod 2$ (i.e., $s_1 z_1 \oplus \cdots \oplus s_n z_n = 0$).

Simon's algorithm: example continued

Perform Hadamard on the first bit of $\frac{1}{\sqrt{2}}(|100\rangle + |111\rangle)$.

$$\frac{1}{2}(|000\rangle - |100\rangle + |011\rangle - |111\rangle)$$

Perform Hadamard on the second bit.

$$\frac{1}{\sqrt{8}}(|000\rangle + |010\rangle - |100\rangle - |110\rangle + |001\rangle - |011\rangle - |101\rangle + |111\rangle)$$

Perform Hadamard on the third bit.

$$\frac{1}{4}(|000\rangle + |001\rangle + |010\rangle + |011\rangle - |100\rangle - |101\rangle - |110\rangle - |111\rangle$$
$$+ |000\rangle - |001\rangle - |010\rangle + |011\rangle - |100\rangle + |101\rangle + |110\rangle - |111\rangle)$$

$$= \frac{1}{2}(|000\rangle + |011\rangle - |100\rangle - |111\rangle)$$

Simon's algorithm: example

Read the three bits. There are four possible answers:

$$000 \qquad 011 \qquad 100 \qquad 111$$

Let $s = s_1 s_2 s_3$.

- 000 gives no information.
- 011 implies $s_2 \oplus s_3 = 0 \mod 2$.
- 100 implies $s_1 = 0 \mod 2$.
- 111 implies $s_1 \oplus s_2 \oplus s_3 = 0 \mod 2$.

Two of the last three equations suffice to determine that $s = 011$ (using that $s \neq 000$).

Simon's algorithm

Measuring the n bits gives a $z \in \{0,1\}^n$ with $\langle s, z\rangle = 0 \bmod 2$, with a uniform probability distribution over all possible solutions z.

$n-1$ linearly independent $z_1, \ldots, z_{n-1} \in \{0,1\}^n$ such that $\langle s, z_i\rangle = 0$ for $i = 1, \ldots, n-1$ suffice to compute s.

The more vectors $z \in \{0,1\}^n$ with $\langle s, z\rangle = 0$ we determine, the larger the probability that $n-1$ of them are linearly independent.

On average, s is determined in polynomial time (in n).

Shor's algorithm (1994)

Problem: Decompose a given $n > 1$ into its prime factors.

Shor's algorithm on average yields a solution in $O(\log^3 n)$ time.

(Note that n is represented using $\log_2 n$ bits.)

Shor's algorithm uses a quantum algorithm to efficiently determine the "period" of a randomly chosen a in $(\mathbb{Z}/n\mathbb{Z})^*$.

With Shor's algorithm, the RSA cryptographic algorithm is broken !

Only, nobody has succeeded yet in building a quantum computer on which Shor's algorithm can be executed for large instances of n...

Quantum circuits versus probabilistic TMs

A probabilistic TM is a nondeterministic TM in which a choice between available transitions can be made with respect to some probability distribution.

Each probabilistic TM can be *efficiently* simulated by a quantum circuit.

Vice versa, each quantum circuit can be simulated by a probabilistic TM.

However, this simulation can take an *exponential* amount of time.

Quantum operations versus probabilistic algorithms

Simon's and Shor's algorithm are *exponentially* faster than all known (classical) probabilistic algorithms for these problems.

Unitary operations on qubits seem to differ in a fundamental way from probabilistic operations on TMs.

"*The only difference between a probabilistic classical world and the equations of the quantum world is that somehow or other it appears as if the probabilities would have to go negative.*"

(Richard Feynman, 1982)

Entanglement also plays a key role.

BQP and BPP

BQP (Bounded error, Quantum, Polynomial time):
Decision problems that can be solved by a quantum TM in polynomial time, with an error probability $< \frac{1}{3}$.

BPP (Bounded error, Probabilistic, Polynomial time):
Decision problems that can be solved by a probabilistic TM in polynomial time, with an error probability $< \frac{1}{3}$.

(The choice for $\frac{1}{3}$ is arbitrary, each value in $\langle 0, 1 \rangle$ would do.)

$$\text{P} \subseteq \text{BPP} \subseteq \text{BQP} \subseteq \text{PSpace}$$

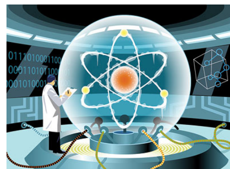
Unknown is whether these inclusions are strict.

The questions BPP,BQP $\subseteq$ NP ? and NP $\subseteq$ BPP,BQP ? are still open.

Quantum computer: dream or reality ?

Building quantum computers is still in its infancy.

A challenge is to avoid that qubits interact with the environment, as else they fall back to a classical state 0 or 1.



Recent breakthroughs:

- At the University of New South Wales a quantum logic gate was built in silicon for the first time, in October 2015.
- D-Wave Systems builds and sells quantum computers which they claim use a 128 qubit processor chipset.