AlDa Script for the Summer Term 2018

Daniel Barley

May 10, 2018

Contents

1 First lecture

Moodle PW: iad18

1.1 Introduction

1.1.1 What is an algorithm?

- an algorithm solves concrete problems and has to be:
  - finite: it has to able to solve the problem with finite resources and in finite steps.
  - terminal: it has to terminate after a finite time.
  - determinism: it has to give the same output when given the same input.
  - analyzable: runtime and memory usage are predictable.
  - correctness: it should be possible to check weather the algorithm solves the problem corectly.

1.1.2 An example

- calculating the factorial:
  - definition $N! = N(N-1)(N-1)\dots 1$
- two possible solutions:
  - loop
  - recursion

```python
def Fak(N):
    for i in range(N + 1):
        fa = fa * i
    return fa
```

```python
def Fak(N):
    if N == 1:
        return 1
    else:
        return N * Fak(N - 1)
```

1.1.3 General workflow:

- Specify the problem
- Create an algorithm
- Look for the most efficient solution
- Convert into elementary steps (implement in a programming language)

1.1.4 Classification of algorithms

- iterative algorithms:
  - sequence of operations is predetermined.
- recursive algorithms:
  - break problem into simpler sub problems
  - special treatment for trivial case.

2 Second Lecture

2.1 Turing Machine

- Proposed by Alan Turing
- Consists of a Control Unit with a write/read head and an internal memory
- the memory is realized by an infinite band of paper
- the instrucctional set consits of
  - read
  - write
  - move to the right (by n cells)
  - move to the left (by n cells)
  - goto a specific cell
  - if `'a'` on memory cell goto k if not goto l
  - stop

2.2 Turings Thesis

- The set of Turing-computable Problems is equivalent to the set of intuitively computable Problems
  - examples of non computable Problems:
    * Halting Problem (check weather Truing code ever halts)
    * Correctness Problem (does a Turing machine compute a desired function)
    * Equivalency Problem (do two Turing Machines compute the same function)

2.3 Analysis of algorithms

A simple example:

- sorting Numbers. Some questions arise:
  - is the algorithm correct
  - how much memory is needed
  - how much time is needed
  - how accurate is the solution, what bandwidth is needed

2.3.1 Correctness

- Induction:
  - A over the naturals $n \in \mathbb{N}$
    - start: show that $A(1)$ is correct
    - step: show that when going from $A(n) \Rightarrow A(n+1)$ it still holds correct
- Invariants
  - loop invariants are constants that do not change in the scope of the loop
- Tests

2.3.2 Complexity

- get the number of needed operations
- Simple description independent from computer and quality of code
- analyze behavior in different cases
  - best case
  - average case
  - worst case

The goal is not to give an exact ETA but to give some guidelines for scalability. Since the average case is hard to guess most analysis consists of best/worst case analysis.

Example

Table 1: Example

n	1	100	1000
$T(n) = n$	1	10	1000
$T(n) = n^2$	1	100	10000
$T(n) = 2^n$	2	$\approx 10^3$	$\approx 10^{30}$

2.4 principles of algorithms

2.4.1 Enumeration

List all the possible solutions.

Example:

- Maximum-Subarray-Problem:
- given the series of reels $a_n, a_1, \dots a_{n-1} \in \mathbb{R}$
- look for consecutive of length R series of elements with maximum partial sum

- example: $31, 41, \underbrace{59, 26, 53, 58, 97}_{187}, 93, 23, 84$

Two simple solution:

```
def maxSubarray(A, n):
    maxsum = 0
    for L in range(n):
        for R in range(L, n):
            sum = 0
            for i in range(L, R):
                sum = sum + A[i]
            if sum > maxsum:
                maxsum = sum
    return maxsum
```

Other solution:

```
def maxSubarray(A, n):
    maxsum = 0
    for L in range(n):
        sum = 0
        for R in range(L, n):
            sum = sum + A[i]
            if sum > maxsum:
                maxsum = sum
    return maxsum
```

2.4.2 Divide and Conquer

- divide the problem in to simpler subproblems until a trivial case is reached
- solve the trivial case
- reconstruct the Solution from solved subproblems

Most of the time the complexity is around $T(n) \approx n \log_2 n$

2.4.3 Scanline-Method

- simple scanning of the array

Example: Isosurfaces: given a lettuce(:)) of points find surfaces by scanning point wise intensity

3 Third Lecture

3.1 simple data structures

- Algorithms
    - efficiency
        * complexity: merit for computing time / memory usage
- Data strucutres
    - efficient access
    - operations on data strucutres
        * need fast algorithms
- Abstract data type
    - specification of what kind of data is stored
    - specification of the kind of operations performed on the structure
- Data strucutres
    - implementation of abstract datatypes
    - maybe different complexities / runtimes

examples:

- arrays
- lists
- stacks
- queue

3.1.1 Arrays

- predefined in most languages
- used for
    - vectors, matrices, images, ...
    - numerics, image processing
- properties
    - selection of congruent elements
    - access via index
- most of the time arrays are statically defined (fixed sized)
- generalization
    - stacks, queues, dynamic arrays

The array as an abstract datatype holds elements of the same datatype. There are some necessary operations an array has to have:

- `create (n)`: creates an array of size `n`
- `A.get(i)`: returns element at position `i`
- `A.set(x, i)`: write `x` to position `i`
- `A.size()`: returns length of array

A dynamic array has to also have the following methods:

- `A.append(x)`: creates a new entry with value `x` at the end
- `A.pop(x)`: deletes the last element

example `vector` of the *C++ standard library* internally allocates, in the simplest case, double the amount of memory. So that at most $k \log n$ accesses are needed, and at most $k \cdot n$ memory is wasted

SIDENOTE: Computer architecture

- CPU
  - program memory: list of operations
  - data storage: data
  - most simple model: sequential computation of operations obtained from the program memory
  - faster: pipelining: divide the operation into smaller parallel instructions like:
    - fetch
    - instruction decode
    - execute
    - mempry
    - write back
  - even faster: multi scalar: local parralelisation
  - now: multicore: multiple CPUs on one die.

good algorithms make good use of architecture.

- Memory
  - hard drive, SSD: good for large amounts of data but slow access
  - memory (DRAM): faster
  - cache hierarchy
    - very fast access, if data is accesses in time wise order
    - third level cahce (SRAM)
    - second level cahce
    - first level cache

good algorithms make good use of memory hierarchy (caching)

- $\boxed{\text{Register}}$
- $\boxed{\text{1st level cache}}$
- $\boxed{\text{2nd level cache}}$
- $\boxed{\text{3rd level cache}}$
- $\boxed{\text{RAM}}$
- $\boxed{\text{Hard drive}}$

3.1.2 lists

- multiple sets of data (maybe of different type)
- access is serial, ideally only operations on neighboring elements
- appending is easy
  - append at the end
  - append as first element
  - middle: append by changing pointers to following elemnts
- cons
  - very slow when not operating serially on neighbors

example:

```python
class listElement:
    def __init__(self,
                 key=0,
                 data=None,
                 next=None,
                 prev=None):
        self.key = key
        self.data = data
        self.next = next
        self.prev = prev
```

different flavours of lists include:

- single chained lists: each element has a reference to its successor
- double chained lsits: each element has a reference to bot its successor and predecessor

example for search with a list:

```python
current = first
while current != None and current_key != x:
    current = current_next
```

is of complexity $\mathcal{O}(n)$

4 Fourth Lecture

4.1 Stacks and Queues

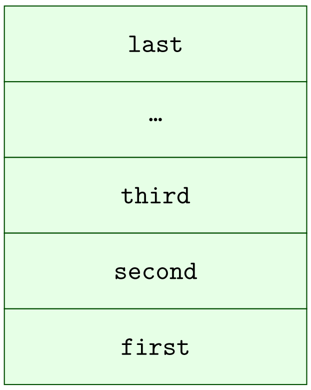
- Stack
  - `create()` creates an empty stack
  - `stack.push(a)` appends element to the ”top” of the stack
  - `stack.pop()` deletes the ”top” element
  - `stack.peek()` looks up the ”top” element, without changing it
  - `stack.is_empty()` returns `true` if the Stack is empty
  - The Stack is a ”LIFO” (Last In First Out) memory
- Queue
  - `queue.enqueue(a)` appends at the ”top”
  - `queue.dequeue()` returns first element and removes it
  - `queue.is_empty()` returns `true` if the Stack is empty
  - The Queue is a ”FIFO” (First In First Out) memory

A special form of the queue is the priority queue which stores key value pairs and supports the following operations

- `pqueue.create()` creates an empty priority queue
- `pqueue.insert(x, key)` inserts a key value pair
- `pqueue.getMin()` returns the element with the smallest key assigned
- `pqueue.deleteMin()` removes the element with the smallest key assigned
- analogous to the `Min` these two methods can be implemented for the largest keys respectively

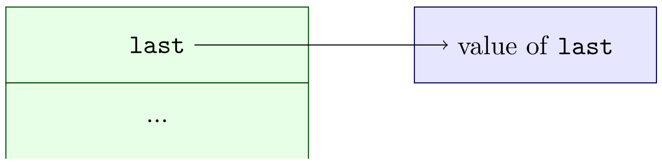
The most common implementation of the priority queue is the heap

4.1.1 Stack/Queue Schematics

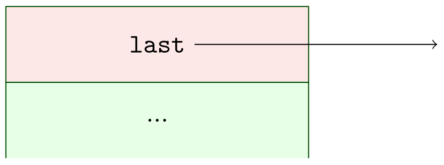


Note that first denotes the first element on the stack/queue time wise ….

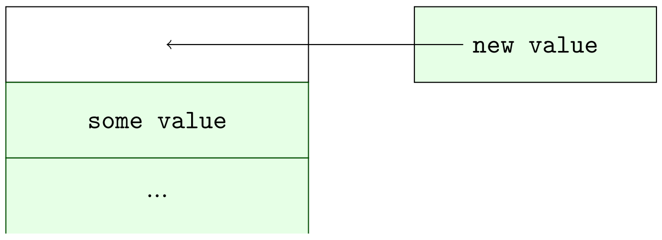
4.1.2 `stack.peek()`



4.1.3 `stack.pop()`



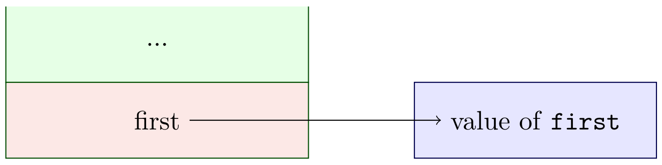
4.1.4 `stack.push()`



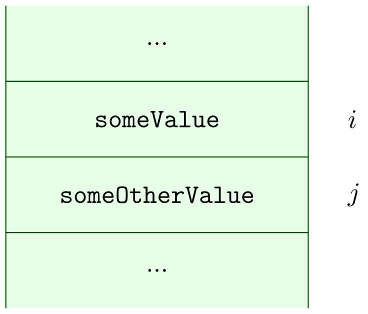
4.1.5 `queue.enqueue`

same as `stack.push()`

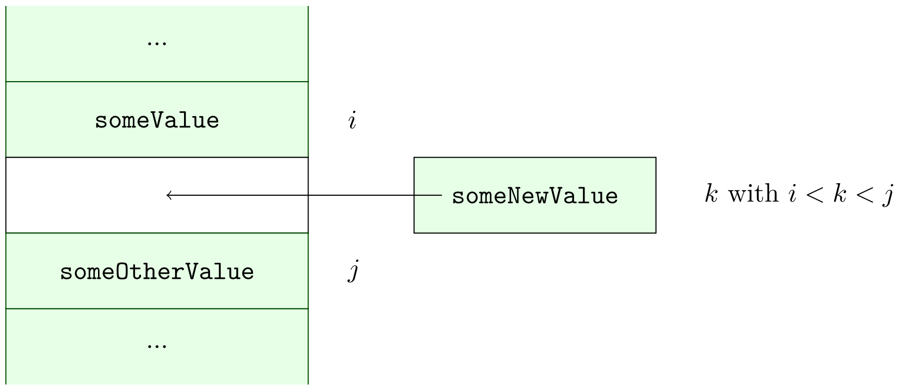
4.1.6 `queue.dequeue`



4.1.7 Priority Queue



4.1.8 `pqueue.insert()`



4.2 Dictionaries (or Maps, Associative Arrays)

A dictionary holds a set of elements each assigned an unambiguous key Instead of an access through an index dictionaries are accessed via the element's key. The following operations are possible:

- `dict.create()` creates an empty dict
- `dict.insert(x, key)` add a value key pair
  - if the key already exists it will be overridden
- `x = dict.find(key)` returns the entry `x`
  - if the key is non existent return a default value
- `dict.delete(key)` removes value belonging to the respective key

Table 2: Complexity

operation	list	array	sorted array	dictionary
`create`	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
`insert`	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n\log n)$
`find`	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$	$\mathcal{O}()$
`delete`	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}()$

4.3 Sorting

4.3.1 Motivation

sorting is one of the most common problems (approx. $\frac{1}{4}$ of the total computing time)

4.3.2 Problem

Given a sequence of n elements $x_1, x_2, \dots, x_n$, the $>$ and $<$ relations have to be defined and $x_i \leq x_j$ should give a boolean (`true/false`). The result should be a sorted sequence i.e $\forall i < j \Rightarrow x_i \leq x_j$. This for example be achieved by permutation (series of swaps).

4.3.3 Selection Sort

The selection sort algorithm is a very simple one, it consists of the following steps:

- find the smallest element
- swap with the first element
- continue

An example implementation:

```python
def selectionSort(alist):
    for i in rage(len(alist) - 1):
        least = i
        for k in range(i + 1, len(alist)):
            if alist[k] < alist[least]:
                least = k
                temp = alist[least]
        alist[least] = alist[i]
        alist[i] = temp
```

4.3.4 Insertion Sort

The insertion sort algorithm is often used by humans when playing card games.

- start with the second element

- check if it is greater than the first
- if so swap the element to the left until smaller than next element
- then continue with the third element
- repeat until sorted

we can divide the array into two parts:

- presorted start
- not yet sorted end

```
def insertionSort(alist):
    for index in range(1, len(alist)):
        current_value = alist[index]
        position = index
        while position > 0 and alist[psition - 1] >
            ↪ current_value:
            alist[position] = alist[position - 1]
        alist[position] = current_value
```

4.4 SIDENOTE: how to record runtime in Python

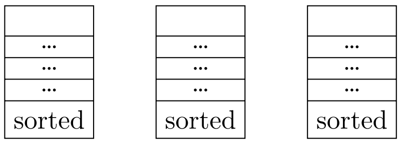
```
    import time

    start_time = time.time()
    run_time = (time.time() - start_time) * 1000 #because
        ↪  time returns microseconds
```

5 Fifth Lecture

5.1 Bubble Sort

- look through array until two values are not sorted, correct that
- repeat until completely sorted
- array is sorted from back to front, i.e first the largest element is pushed to the back

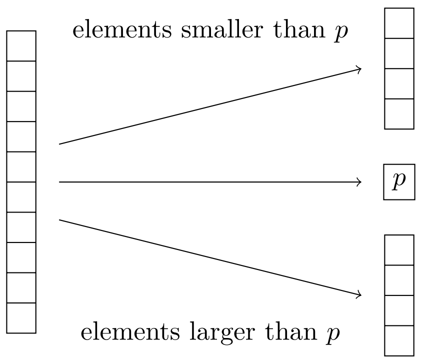


```
def bubbleSort(nlist):
    for j in range(len(nlist) -1):
        for i i range(nlist-i-j):
            if nlist[i] > nlist[i+1]:
                temp = nlist[i]
                nlist[i] = nlist[i+1]
                nlist[i+1] = temp
```

5.2 Divide and Conquer

5.2.1 Quick Sort

- choose an (ideally median) element from list as pivot
- create two lists and sort in elements accordingly
  - left: smaller than pivot
  - right: larger than pivot
  - sort both sub lists



- divide problem into (two or more) sub problems
- recursively solve those sub problems
- combine sub solutions to solve the initial problem

How to pivot the array:

- go through array from left to right
- if the element's index and value are smaller than the pivot do nothing
- if not swap with an element from the other side
- take l and r to walk through the array from left to right (l) or from right to left (r)
- increment l until $A[\ l\ ] > p$ (this element has to be swapped to the right)
- decrement r until $A[\ r\ ] \leq p$ (this element has to be swapped)
- swap element l with r
- $l = l + 1$ and $r = r - 1$
- done when $l = r$

Note that choice matters when it comes to the pivot. Ideally the pivot splits the list into two parts, almost equal in size. But Since calculating the median requires sorting simply choosing the median is not an option. In reality some random element is often chosen as pivot, like the first or last element or a random choice. For more precision an assortment of elements can randomly be selected from the list. The pivot is then obtained by choosing the median of those elements.

Even though quick sort is somewhat fast (of average complexity $\mathcal{O}(n \cdot \log n)$) it has one problem. Choosing an inefficient pivot (deviating a lot from the median) adds a lot of complexity.

5.2.2 Merge Sort

Merge Sort avoids quick sort's pivot problems by not using one.

- divide list in the middle
- recursively sort the sub lists
- combine sorted sub lists

6 Sixth Lecture

6.1 Merge Sort revisited

Table 3: Quicksort vs. Mergesort

Quicksort	Mergesort
$\mathcal{O}(n \cdot \log n)$ on average $\mathcal{O}(n^2)$ in worst case	$\mathcal{O}(n \cdot \log n)$ on average $\mathcal{O}(n \cdot \log n)$ in worst case

```
MergeSort(A, start, end, tmp):
    if end - start > 1:
        middle = start + (end - start) // 2 #integer
           ↪ division
        MergeSort(A, start, middle, tmp)
        MergeSort(A, middle, end, tmp)
    pos = start
    i = start
    j = middle
    while pos < end:
        if(i < middle) and (j = end) or (A[i] < A[j]):
            tmp[pos] = A[i]
            i += 1
            pos += 1
        else:
            tmp[pos] + A[j]
            j += 1
            pos += 1
```

So merge Sort is guaranteed to be of complexity $n \cdot \log n$.

6.2 Complexity

- How does the computing time change with respect to the problem's size?
- different possible approaches for merit
  - simply measure computing time
    - depends on the used hardware
    - depends on implementation and programming language
  - count the number of elementary operations
    - better approximation
    - still deviations on real systems (caching)

The complexity is an abstract merit for runtime predictions. A naive approach would be to simply count the number of atomic operation needed to complete the task at hand. For example:

- $+, -, /, *, \%$
- memory access
- function calls

For this to be consistent some approximations have to be made:

- all atomic operations cost exactly one unit of time
- all memory access is equally fast

6.2.1 example: Insertion Sort

Table 4: Atomic Operations Insertion Sort

insertionSort(A)	cost	number of calls
`for j=2 to A.lenght`	C_1	n
`key = A[i]`	C_2	$n-1$
`i = j-1`	C_4	$n-1$
`while j > 0 and a[i] > key`	C_5	$\sum_{j=2}^{n} t$
`    A[i+i] = A[i]`	C_6	$\sum_{j=2}^{n}(t_j - 1)$
`    i = j - 1`	C_7	$\sum_{j=2}^{n}(t_i - 1)$
`A[i + j] = key`	C_8	$n-1$

6.2.2 Further Simplification

Example:

- Runtime is given by $T(n) = an^2 + bn + c$
- The parameters a, b, c still depend on the implementation and the hardware

In order to obtain a hardware independent estimate we drop all parameter and only take the fastest growing term into account.

- up until now: look at the number of atomic operations
    - simplification 1: calculate upper limit by counting atomic operations
    - Simplification 2: parameter is size of input n

6.3 Bachmann-Landau Notation or Big-O Notation

Using the Big-O notation the asymptotic behavior of functions can be approximated.

- $\exists C$ so that $C : T(n) \leq Cf(n) \Rightarrow T(n) \in \mathcal{O}(f(n))$
- $\exists C'$ so that $C'T(n) \geq C'f(n) \Rightarrow T(n) \in \Omega(f(n))$

formal definitions:

- $\mathcal{O}(f(n)) = \{f \mid \exists C > 0, n_0 > 0 \; \forall n \geq n_0 : f(n) \leq g(n)\}$
- $f(n) \in \mathcal{O}(g(n))$, if there exists constant $C > 0, n_0 > 0$ so that $f(n) \leq Cg(n)$ for all $n \geq n_0$
- $\Omega(f(n)) = \{f \mid \exists C > 0, n_0 > 0 \; \forall n \geq n_0 : f(n) \geq g(n)\}$
- $f(n) \in \mathcal{O}(g(n))$, if there exists constant $C > 0, n_0 > 0$ so that $f(n) \geq Cg(n)$ for all $n \geq n_0$
- in short:
- $f(n) \in \mathcal{O}(g(n))$
    - $f(n)$ does not grow faster than $g(n)$ (asymptotically)
- $f(n) \in \Omega(g(n))$
    - $f(n)$ does grow at least as fast as $g(n)$ (asymptotically)

6.3.1 Example Bubble Sort

Pseudo-Code:

```
BubbleSort(A):
    for i in range(0, n-2):
        for j in range(0, n-2-i):
            if A[j] > A[i+j]:
                swap(A[j], A[j+1])
```

$T(n) \leq c \underbrace{\#\text{loop count}}_{X(n)}$ so $c'n^2 \leq X(n) = \sum_{i=0}^{n-2}(n-2-i) \leq cn^2$ therefore

$T(n) \in \mathcal{O}(n^2)$ with $T(n) = \begin{cases} f(n) \in \Omega(n^2) \\ f(n) \in \Theta(n^2) \end{cases}$