FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG
NATURWISSENSCHAFTLICHE FAKULTÄT

Background on Algorithms, Intro Supervised Learning

Lecture "Mathematical Data Science" 2021/2022

Frauke Liers
Friedrich-Alexander-Universität Erlangen-Nürnberg
3.11.2021

DISCRETE OPTIMIZATION

Excurus: Some Background in Algorithms

this excursus is meant to give some brief and abstract introduction into algorithms. further reading, e.g.:

- Thomas Cormen, Charles Leiserson, Ronald Rivest, and Cliff Stein: Introduction to Algorithms, MIT Press
- Thomas Ottmann and Peter Widmayer: Algorithmen und Datenstrukturen,
- Robert Sedgewick, Kevin Wayne: Algorithms, Addison-Wesley,
- Donald Knuth: The Art of Computer Programming,

and many others.
for exemplary purposes, we do it for a basic operation, namely sorting numbers.

Sorting: Notation

basic operation in computer science

- **Input:** n numbers $\langle a_1, a_2, \ldots, a_n \rangle$.
- **Output:** permutation $\langle a'_1, a'_2, \ldots, a'_n \rangle$ such that $a'_1 \leq a'_2 \leq \ldots \leq a'_n$.

Sorting: Notation

basic operation in computer science

- **Input:** n numbers $\langle a_1, a_2, \ldots, a_n \rangle$.
- **Output:** permutation $\langle a'_1, a'_2, \ldots, a'_n \rangle$ such that $a'_1 \leq a'_2 \leq \ldots \leq a'_n$.
- **instance**: each concrete series of numbers, e.g.,
  $\langle 32, 25, 13, 48, 39 \rangle \Rightarrow \langle 13, 25, 32, 39, 48 \rangle$
  (in general: all input that is necessary for determining a solution.)

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG
NATURWISSENSCHAFTLICHE FAKULTÄT

Sorting: Notation

basic operation in computer science

- **Input:** n numbers $\langle a_1, a_2, \ldots, a_n \rangle$.
- **Output:** permutation $\langle a'_1, a'_2, \ldots, a'_n \rangle$ such that $a'_1 \leq a'_2 \leq \ldots \leq a'_n$.
- **instance**: each concrete series of numbers, e.g., $\langle 32, 25, 13, 48, 39 \rangle \Rightarrow \langle 13, 25, 32, 39, 48 \rangle$ (in general: all input that is necessary for determining a solution.)

Depending on the application, different algorithms are suited best:

- How many elements?
- Is already partially sorted?
- In main memory, or on disk?

$\vdots$

Sorting: Notation

basic operation in computer science

- **Input:** n numbers $\langle a_1, a_2, \ldots, a_n \rangle$.
- **Output:** permutation $\langle a'_1, a'_2, \ldots, a'_n \rangle$ such that $a'_1 \leq a'_2 \leq \ldots \leq a'_n$.
- **instance**: each concrete series of numbers, e.g.,
  $\langle 32, 25, 13, 48, 39 \rangle \Rightarrow \langle 13, 25, 32, 39, 48 \rangle$
  (in general: all input that is necessary for determining a solution.)

Depending on the application, different algorithms are suited best:

- How many elements?
- Is already partially sorted?
- In main memory, or on disk?

⋮

An algorithm is called **correct**, if it **terminates** for all instances with a correct solution. It then **solves** the problem.

Insertion Sort

Figure 2.1 Sorting a hand of cards using insertion sort.

Aus: Cormen et al. (2001) "Algorithms", chpt. 2;
MIT Press, Cambridge (MA)

Insertion Sort

As parameters, it has the array `A` and its length `length(A)`. In the for-loop, the j-th element of the sequence is inserted in the correct position that is determined by the while-loop. In the latter we compare the element to be inserted (`key`) from 'right' to 'left' with each element from the sorted subsequence stored in `A[0],...,A[j-1]`. If `key` is smaller, it has to be insert further left. Therefore, we move `A[i]` one position to the right and decrease `i` by one in line 7. If the while-loop stops, `key` is inserted.

```
insertion_sort(A)
  for j = 1 to (length(A)-1) do
    key = A[j]
    // insert A[j] into the sorted sequence A[1...j-1]
    i = j
    while (i > 0 and (A[i-1] > key) ) do
      A[i] = A[i-1]
      i = i-1
    end while
    A[i] = key
    end for
```

algorithm for sequence $\langle 5, 2, 4, 6, 1, 3 \rangle$

A[1]	**A[2]**	**A[3]**	**A[4]**	**A[5]**	**A[6]**
5	2	4	6	1	3
2	5	4	6	1	3
2	4	5	6	1	3
2	4	5	6	1	3
1	2	4	5	6	3
1	2	3	4	5	6

Correctness

- at beginning of `for`-loop, `A[1..j-1]` always sorted: within `for`-loop, `A[j-2]`, `A[j-3]`, `A[j-4]`, ... are moved one position to the right until the correct position for `key=A[j]` is found and assigned to `key`.

Correctness

- at beginning of `for`-loop, `A[1..j-1]` always sorted: within `for`-loop, `A[j-2]`, `A[j-3]`, `A[j-4]`, ... are moved one position to the right until the correct position for `key=A[j]` is found and assigned to `key`.
- when `for`-loop stops, it is `j=n` and `A[0..n-1]` is sorted.

Correctness

- at beginning of `for`-loop, `A[1..j-1]` always sorted: within `for`-loop, `A[j-2]`, `A[j-3]`, `A[j-4]`, ... are moved one position to the right until the correct position for `key=A[j]` is found and assigned to `key`.
- when `for`-loop stops, it is `j=n` and `A[0..n-1]` is sorted.
- termination: `while`- and `for`-loop always terminate.

Used Ressources

- space
- **time**

Used Ressources

- space
- **time**

RAM: “random access machine”:

- 1 processor
- data in main memory
- All memory accesses need the same time.

Used Ressources

- space
- **time**

RAM: “random access machine”:

- 1 processor
- data in main memory
- All memory accesses need the same time.

time:

- number of primitive operations, e.g., $a = b$, $a = b + c$, etc.

Used Ressources

- space
- **time**

RAM: "random access machine":

- 1 processor
- data in main memory
- All memory accesses need the same time.

time:

- number of primitive operations, e.g., $a = b$, $a = b + c$, etc.
- assumption: each such operation takes equal time.

Used Ressources

- space
- **time**

RAM: "random access machine":

- 1 processor
- data in main memory
- All memory accesses need the same time.

time:

- number of primitive operations, e.g., $a = b$, $a = b + c$, etc.
- assumption: each such operation takes equal time.

“Worst Case”-Running Time

- **longest possible running time** for predescribed input size (here n).

“Worst Case”-Running Time

- **longest possible running time** for predescribed input size (here n).
- **upper bound** for running time for solving an arbitrary instance of this size.

“Worst Case”-Running Time

- **longest possible running time** for predescribed input size (here n).
- **upper bound** for running time for solving an arbitrary instance of this size.
- For some applications, “worst-case” appears more often than for others, e.g., when searching for a non-existent entry in a data base.

"Worst Case"-Running Time

- **longest possible running time** for predescribed input size (here n).
- **upper bound** for running time for solving an arbitrary instance of this size.
- For some applications, "worst-case" appears more often than for others, e.g., when searching for a non-existent entry in a data base.

Question: Does the worst-case running time of insertion_sort grow linearly, quadratically, . . ., or even exponentially in n?

Running Time Insertion Sort

```
void insertion_sort(A){
  for j = 1 to < length(A-1) do
    key = A[j]
    // insert A[j] into the sorted sequence A[1...j-1]
    i = j
    while (i > 0 and A[i-1] > key) do
      A[i] = A[i-1]
      i= i-1
    end while
    A[i] = key
  end for
```

Running Time Insertion Sort

```
void insertion_sort(A){
  for j = 1 to < length(A-1) do
    key = A[j]
    // insert A[j] into the sorted sequence A[1...j-1]
    i = j
    while (i > 0 and A[i-1] > key) do
      A[i] = A[i-1]
      i= i-1
    end while
    A[i] = key
  end for
```

for sequence sorted in reverse order (worst case)

- while-loop stops only when $i = 0$
- $j = 1$: 1 assignment $A[i] = A[i-1]$
- $j = 2$: 2 assignments
- $\ldots$
- $j = n-1$: $n-1$ assignments

Running Time Insertion Sort

```
void insertion_sort(A){
  for j = 1 to < length(A-1) do
    key = A[j]
    // insert A[j] into the sorted sequence A[1...j-1]
    i = j
    while (i > 0 and A[i-1] > key) do
      A[i] = A[i-1]
      i= i-1
    end while
    A[i] = key
  end for
```

for sequence sorted in reverse order (worst case)

- while-loop stops only when $i = 0$
- $j = 1$: 1 assignment $A[i] = A[i-1]$
- $j = 2$: 2 assignments
- $\ldots$
- $j = n-1$: $n-1$ assignments

in total: $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$ many assignments; quadratically many

...more formally

Worst-Case Running Time

Consider only the characteristic behavior as a function of the input size; ignore constants and terms of lower order.

...more formally

Worst-Case Running Time

Consider only the characteristic behavior as a function of the input size; ignore constants and terms of lower order.
If the values of f are above some function g for all n larger than some constant n_0, then asymptotically f is an upper bound for g, notation $g = O(f)$.

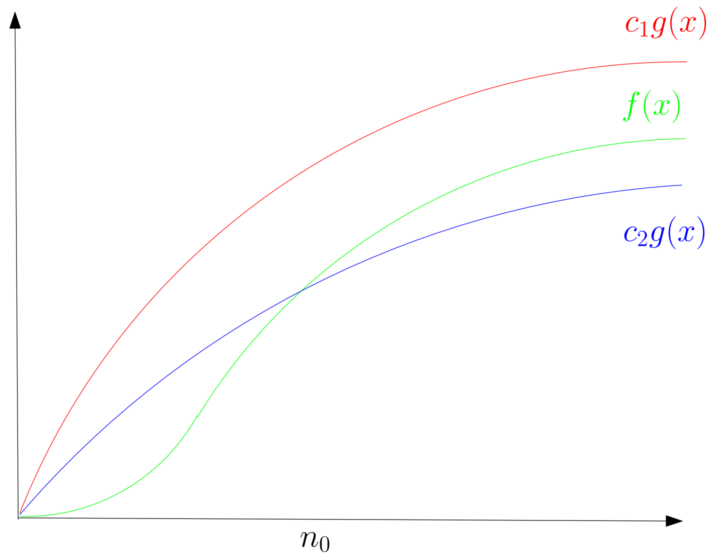
...more formally

Worst-Case Running Time

Consider only the characteristic behavior as a function of the input size; ignore constants and terms of lower order.
If the values of f are above some function g for all n larger than some constant n_0, then asymptotically f is an upper bound for g, notation $g = O(f)$.

$$
\begin{aligned}
O(g(n)) &:= \{f(n) | (\exists c, n_0 > 0)(\forall n \geq n_0) : 0 \leq f(n) \leq cg(n)\} \\
\Theta(g(n)) &:= \{f(n) | (\exists c_1, c_2, n_0 > 0)(\forall n \geq n_0) : c_1 g(n) \leq f(n) \leq c_2 g(n)\} \\
\Omega(g(n)) &:= \{f(n) | (\exists c, n_0 > 0)(\forall n \geq n_0) : 0 \leq cg(n) \leq f(n)\}
\end{aligned}
$$

Worst-Case Running Time



- The *worst case* running time of insertion sort is $O(n^2)$.

Design of Algorithms

insertion sort: *incremental method.*
different principle: *‘divide and conquer’*

- **divide** problem in subproblems
- **conquer** the subproblems through recursive solution. (If small enough, solve them directly.)
- **combine** the solutions of the subproblems to a solution for the original problem.

Merge Sort

- **divide:** divide sequence of n numbers in the middle into two sub sequences.
- **conquer:** sort the subsequences recursively using *merge sort*.
- **combine:** merge the two subsequences to a sorted sequence.

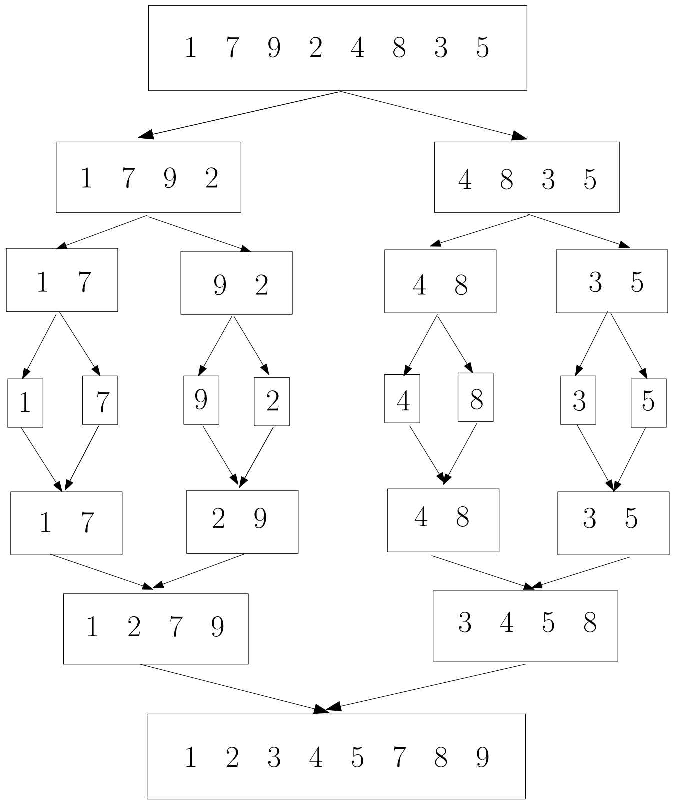
$\longrightarrow$ for sequences containing one element only nothing has to be done.

merge sort

sort sequence stored in `A[p]...A[r]`

```
void merge_sort(int[] A, int p, int r) {
  int q; /* Middle of the sequence */
  if (p < r) {  /* if p = r: only 1 element */
    q = p+((r-p)/2);
    merge_sort (A, p, q);  /* left subsequence */
    merge_sort (A, q+1, r);  /* right subsequence */
    merge (A, p, q, r);  /* merge subsequences */
  }
}
```

Illustration Merge Sort



Merge Sort

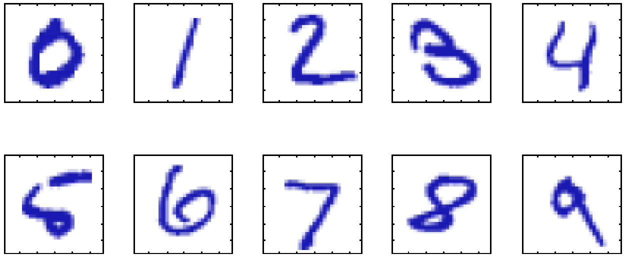
It can be shown: worst-case running time of merge sort is $O(n \log n)$ (better than insertion sort)
In fact: any algorithm for sorting n numbers that uses only comparisons and moves of numbers needs at least $\Omega(n \log n)$.
BTW: try the shell-command **sort**!

End of Excursus

Introduction Supervised Learning

(see Bishop Pattern Recognition book, chapter 1)
...back to our lecture topics: now supervised learning:

Figure 1.1 Examples of hand-written digits taken from US zip codes.



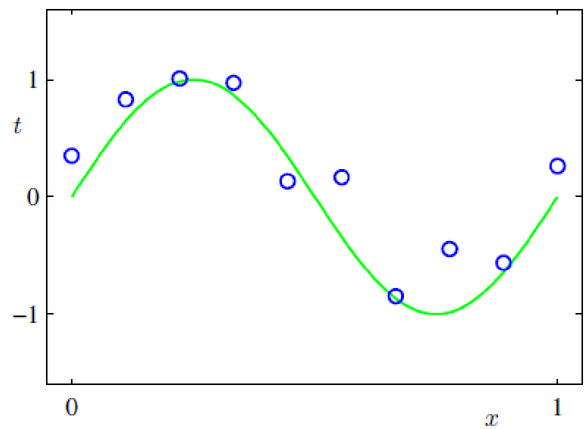
- large *training* data set with N points $\{x_1, \ldots, x_N\}$
- labels / categories (e.g., digits in handwriting, different objects...) are known in advance ('labeled data'), stored in vector t ('target') for each data point
- use training data for tuning parameters of an adaptive model
- *training phase or learning phase* results in function $y(x)$ that takes data point x as input, returns $y(x)$ that corresponds to a specific target / label
- *generalization*: additional data contained in *test set* can be used to categorize new data is main step

Introduction Supervised Learning

- *feature extraction*: reduce difficulty of the problem by reduction of data through preprocessing (scaling,...) and/or dimensionality reduction
- digit recognition is *classification problem*

Regression: Polynomial Curve Fitting

Figure 1.2 Plot of a training data set of $N = 10$ points, shown as blue circles, each comprising an observation of the input variable x along with the corresponding target variable t. The green curve shows the function $\sin(2\pi x)$ used to generate the data. Our goal is to predict the value of t for some new value of x, without knowledge of the green curve.



- use data x to predict value of real target value t
- assume there is some underlying regularity, i.e., there is something to 'learn'
- given: training set $\mathbf{x} = (x_1, \ldots, x_N)$, $\mathbf{t} = (t_1, \ldots, t_N)$

Regression: Polynomial Curve Fitting

- fit data using polynomial funktion of form

$$y(x, \mathbf{w}) = w_0 + w_1 x + w_2 x^2 + \ldots + w_M x^M = \sum_{j=0}^{M} w_j x^j$$

  M order ot polynomial, real coefficients w_j
- $y(x, \mathbf{w})$ is a polynomial, but only *linear* in unknown coefficients w that are determined by fitting polynomial to training data
- minimize *error function* between data and prediction, e.g. $E(\mathbf{w}) = \frac{1}{2} \sum_{n=1}^{N} (y(x_n, \mathbf{w}) - t_n)^2$
- error minimization: gradient w.r.t. w is linear function with unique solution $\mathbf{w}^\star$, yields solution $y(x, \mathbf{w}^\star)$.
- choice of M important

Regression: Polynomial Curve Fitting

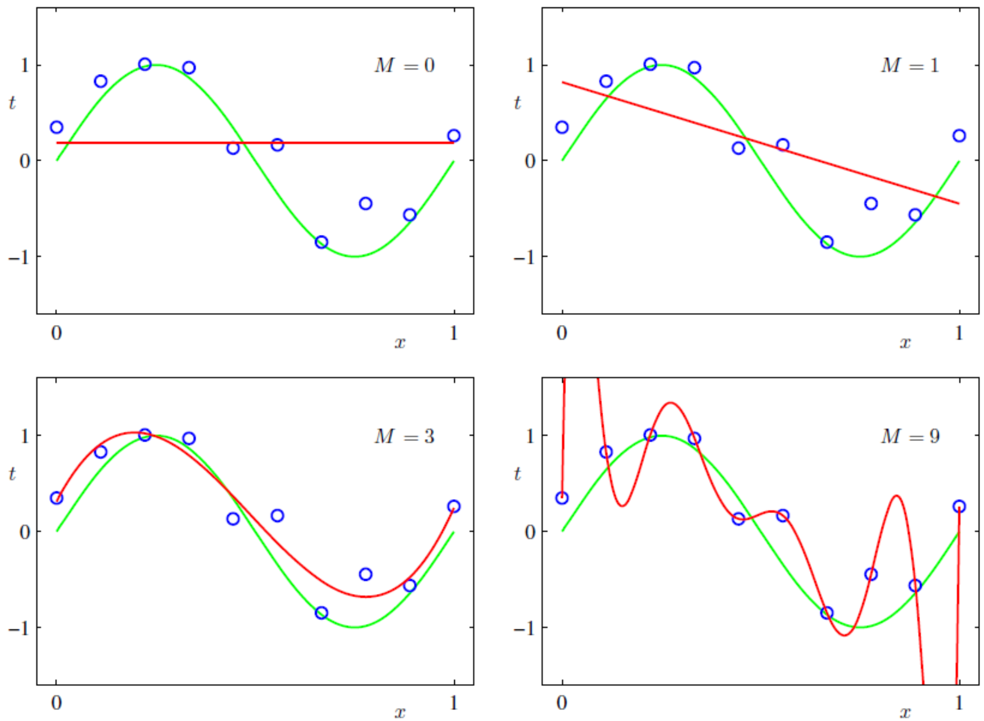
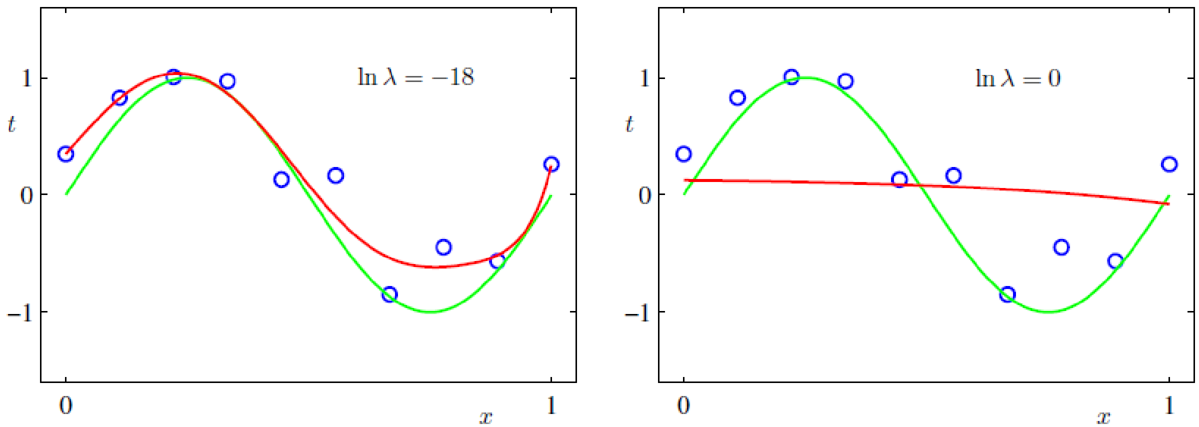


Figure 1.4 Plots of polynomials having various orders M, shown as red curves, fitted to the data set shown in Figure 1.2.

problem: $M = 3$ is good, but too large M yields zero error, but *overfitting*: poor representation of underlying cosine function, poor generalization.
underlying problem for large M: large (positive and negative) coefficients

Avoid Overfitting by Regularization

- add a penalty term in order to avoid large coefficients.
  $\tilde{E}(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N}(y(x_n, \mathbf{w}) - t_n)^2 + \frac{\lambda}{2}||\mathbf{w}||^2$
- λ governs importance of regularization
- can still be minimized in closed form



igure 1.7 Plots of $M = 9$ polynomials fitted to the data set shown in Figure 1.2 using the regularized error ınction (1.4) for two values of the regularization parameter λ corresponding to $\ln \lambda = -18$ and $\ln \lambda = 0$. The ase of no regularizer, i.e., $\lambda = 0$, corresponding to $\ln \lambda = -\infty$, is shown at the bottom right of Figure 1.4.