ISyE 6416: Computational Statistics
Spring 2017

Lecture 2: Aspects of algorithms

Prof. Yao Xie

H. Milton Stewart School of Industrial and Systems Engineering
Georgia Institute of Technology

Top 10 Algorithms of the 20th Century

1. 1946: The Metropolis Algorithm for Monte Carlo. Through the use of random processes, this algorithm offers an efficient way to stumble toward answers to problems that are too complicated to solve exactly.
2. 1947: Simplex Method for Linear Programming. An elegant solution to a common problem in planning and decision-making.
3. 1950: Krylov Subspace Iteration Method. A technique for rapidly solving the linear equations that abound in scientific computation.
4. 1951: The Decompositional Approach to Matrix Computations. A suite of techniques for numerical linear algebra.
5. 1957: The Fortran Optimizing Compiler. Turns high-level code into efficient computer-readable code.
6. 1959: QR Algorithm for Computing Eigenvalues. Another crucial matrix operation made swift and practical.
7. 1962: **Quicksort Algorithms for Sorting**. For the efficient handling of large databases.
8. 1965: **Fast Fourier Transform**. Perhaps the most ubiquitous algorithm in use today, it breaks down waveforms (like sound) into periodic components.
9. 1977: Integer Relation Detection. A fast method for spotting simple equations satisfied by collections of seemingly unrelated numbers.
10. 1987: Fast Multipole Method. A breakthrough in dealing with the complexity of n-body calculations, applied in problems ranging from celestial mechanics to protein folding.

`https://www.siam.org/pdf/news/637.pdf`

- Aspects of algorithms
- Close-look at bisection, quick-sort, and FFT

Aspects of computer algorithms

Two most important aspects of algorithms

- Accuracy: e.g., absolute error $|\hat{\theta} - \theta|$ or relative error $|\hat{\theta} - \theta|/|\theta|$
- Efficiency: how many computing **resources** are needed to achieve certain **precision**

Other aspects

- Robustness
- Stable

Algorithm accuracy

- Some algorithms are exact (e.g., multiplying two matrices)
- Other algorithms are approximate because the result to be computed does not have finite closed-form solution
  - Truncation error: solving $f(x) = 0$ by solving $\tilde{f}(x) = 0$, where $\tilde{f}(x)$ is the Taylor expansion of $f(x)$
  - Discretization error: solving continuous PDE via discretization $\partial f(x)/\partial x = g(x)$
  - Approximation error: solve $f(x) = 0$ using gradient descent.

Algorithm and data

- Performance of algorithm may depend on the data

Example [Wilkinson 59]

$f(x) = (x-1)(x-2)\cdots(x-20) = x^{20} - 210x^{19} + \cdots + 20!$
if 210 is changed to $210 + 2^{-23}$ the root may change drastically

"condition of data": quantify the condition of a set of data for a particular set of operations

- finding root: increase in $1/\dot{f}(x)$ near the root
- condition number of a matrix M:
  $\kappa(M) = \lambda_{\max}(M)/\lambda_{\min}(M)$

Algorithm robustness

An algorithm is robust if it can be applied reliably to a wide range of data.

Example: robust optimization

Robust linear programming addresses linear programming problems where the data is uncertain, and a solution which remains feasible despite that uncertainty is sought.

$$\text{(LP)} \quad \min_x \; c^\intercal x : \quad a_i^\intercal x \le b_i, \quad i = 1, \ldots, m.$$

$$\text{(Robust LP)} \quad \min_x \; c^\intercal x : \quad a_i \in \mathcal{U}_i, a_i^\intercal x \le b_i, \quad i = 1, \ldots, m.$$

Robust Optimization, 2009. Aharon Ben-Tal, Laurent El Ghaoui & Arkadi Nemirovski.

Algorithm stability

- If the algorithm always yields to a solution that is an exact solution to a perturbed problem $\tilde{f}(x) = f(x + \delta x)$, the algorithm is said to be **stable**.
- A small perturbation will not result in big difference in solution of a stable algorithm.
- Perturbation to input data may be due to truncation error
- *if problem is ill-conditioned, even stable algorithm may produce bad results.*

Algorithm efficiency

- usual measure of efficiency is speed, i.e., how long an algorithm takes to produce its result
- analysis of algorithm: to determine the amount of resources (time and storage) needed to execute an algorithm
- complexity of algorithm:
  - time complexity: count the # of operations (flops)
  - worst-case running time: the longest running time for any input of size n
  - order of growth of the running time that really interests us asymptotic analysis and big O notation
    e.g. $f(n) = 9 \log n + 5(\log n)^3 + 3n^2 + 2n^3 = \mathcal{O}(n^3)$, as $n \to \infty$
  - also care of "memory" complexity

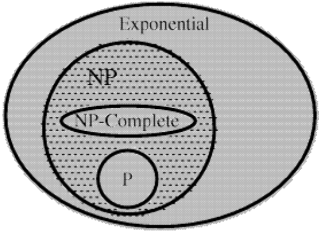
P vs. NP

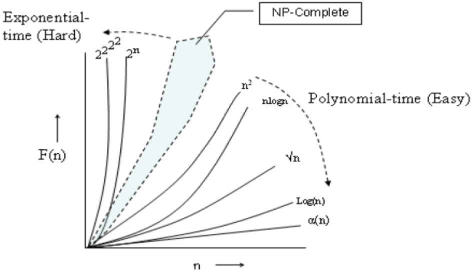
NP: non-deterministic polynomial

- polynomial-time algorithms: on inputs of size n, the worst-case running time is $\mathcal{O}(n^k)$ for some constant k: e.g., sorting
- there are also problems that can be solved but not in time $\mathcal{O}(n^k)$ for any constant k
- we think of problems that are solvable by polynomial-time algorithms as being tractable, or easy, and problems that require superpolynomial time as being intractable, or hard

NP complete

- an interesting class of problems, called the **NP-complete** problems, whose status is unknown: no polynomial-time algorithm has yet been discovered for an NP-complete problem, nor has anyone yet been able to prove that no polynomial-time algorithm can exist for any one of them
- NP-complete problems: finding clique, travel salesman





Common approaches to design algorithm

- recursion: an algorithm that recursively calls itself.
  Example: compute mean and variance

Horner's method

$$p_d(x) = c_d x^d + \cdots + c_1 x + c_0$$

evaluated as

$$p_d(x) = x(\cdots x(x(c_d x + c_{d-1}) + \cdots) + c_1) + c_0$$

- divide and conquer
  A problem is broken into subproblems, each of which is solved, and then the subproblem solutions are combined into a solution for the original problem.
  Example: "bubble sort" versus **quick sort**, **FFT**.

- greedy algorithm
  - Each step is as efficient as possible **without** regarding future steps
  - Greedy algorithm is usually used in the early stages of computation for a problem or when a problem lacks understandable structures.
  - Example: **gradient descent**, **Newton's method**.
- Iterative method: **bisection**
  Convergence: whether or not it will ends with a (right) fix point if iterate enough steps
- Convex relaxations
  replace non-convex constraints with convex ones.

Example: convex relaxation for variable selection

Solve $y = Ax$ where $x \in \mathbb{R}^n$ is k-sparse.

Exponential complexity

$$\begin{array}{rl} \text{minimize} & \|x\|_0 \\ \text{subject to} & y = Ax \end{array}$$

Polynomial complexity

$$\begin{array}{rl} \text{minimize} & \|x\|_1 \\ \text{subject to} & y = Ax \end{array}$$

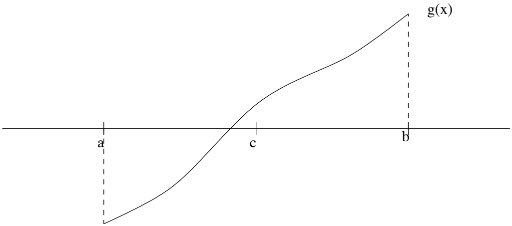
lasso algorithm and compressed sensing

Next, we exam 3 instances closely

- Bisection
- Quicksort
- Convolution and FFT

Bisection

- for a **continuous monotone** function $g(x)$, find root such that
$$x^* : g(x^*) = 0$$
- bisection:
  - start with $g(a) < 0 < g(b)$
  - take $c = \frac{1}{2}(a + b)$
  - if $g(c) < 0$, consider right half interval $[c, b]$
  - if $g(c) > 0$, consider left half interval $[a, c]$
  - repeat the above corresponding subinterval



c=1/2(a+b)

- By doing this, we always have

$$g(x_L) < 0 < g(x_r)$$

- since g is continuous, there must be a point $x^* \in [x_l, x_r]$ such that $g(x^*) = 0$
- the length of the interval is halved each time
- after n iterations, the final bracketing interval has length $2^{-n}(b-a)$, this means

$$|x^* - x| < 2^{-n}(b-a), \quad \forall x \in [a,b]$$

- the length of the interval converges to 0 as $n \to \infty$ (convergence rate $e^{-n\log 2}$: exponential convergence rate)

Quicksort algorithm

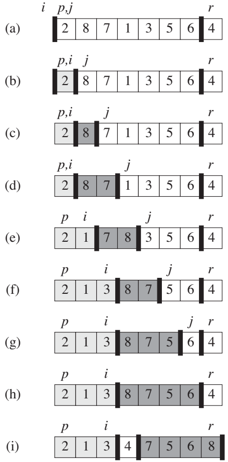
- sorting: $\{3, 1, 5, 2\} \Rightarrow \{1, 2, 3, 5\}$
- Native "bubble sort"
  Starting from the beginning of the list, compare every adjacent pair, swap their position if they are not in the right order (the latter one is smaller than the former one). After each iteration, one less element (the last one) is needed to be compared until there are no more elements left to be compared.
- complexity:$\mathcal{O}(n^2)$

- Quicksort: a recursive algorithm using “divide-and-conquer”
- a vector of numbers c of length n, start location for sort p, end location for sort q
- peudocode

```
quicksort(c, p, q)
r := findpivot(c, p, q)
quicksort(c, p, r-1)
quicksort(c, r+1, q)
```

Demo: http://me.dt.in.th/page/Quicksort/

example of
PARTITION
(to find pivot)



- best scenario: [1, 2, 3, 4, 5, 6, 7, 8], examining about $n \log n$ elements
- worst scenario: [8, 7, 6, 5, 4, 3, 2, 1]
  pass 1: [1, 8, 7, 6, 5, 4, 3, 2]
  pass 2: [1, 2, 8, 7, 6, 5, 4, 3]
  pass 3: [1, 2, 3, 8, 7, 6, 5, 4]
  ...

Complexity of quicksort

Average complexity of quicksort algorithm is $\mathcal{O}(n \log n)$.

Proof:
Homework: show that the worst-case complexity of quicksort is $\mathcal{O}(n^2)$.

Convolution

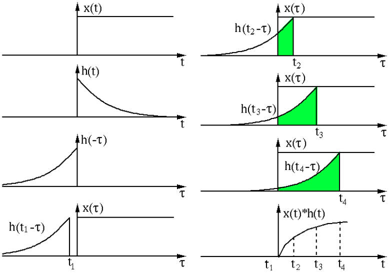
- convolution of two functions

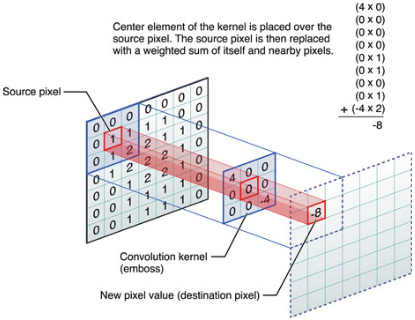
$$\text{continuous: } R(t) = x(t) \star h(t) = \int x(t-u)h(u)du$$

$$\text{discrete: } R[m] = x \star h = \sum_j x_{m-j} h_j$$

- very important in statistics, image processing, signal processing, computer science.
  - distribution of sum of two random variables $U + V$ is the convolution of their PDFs
  - "filtering" of an image

(Section 3, "Computational statistics" by J. Gentle.)


x(t)
h(t)
h(-τ)
h(t1 -τ)
x(τ)
h(t2-τ)
h(t3-τ)
h(t4-τ)
x(t)*h(t)
t1
t2
t3
t4
τ
t


Center element of the kernel is placed over the source pixel. The source pixel is then replaced with a weighted sum of itself and nearby pixels.
(4 x 0)
(0 x 0)
(0 x 0)
(0 x 0)
(0 x 1)
(0 x 1)
(0 x 0)
(0 x 1)
+ (-4 x 2)
-8
Source pixel
Convolution kernel
(emboss)
New pixel value (destination pixel)

Convolution and Fourier transform

- Fourier transform, denoted as $\mathcal{F}(x)$ is defined as

$$\mathcal{F}(x) \triangleq X(f) = \int_{-\infty}^{\infty} x(t) e^{-i2\pi f t} dt$$

where $i = \sqrt{-1}$

- Inverse Fourier transform

$$x(t) = \int_{-\infty}^{\infty} X(f) e^{i2\pi f t} df$$

Convolution theorem

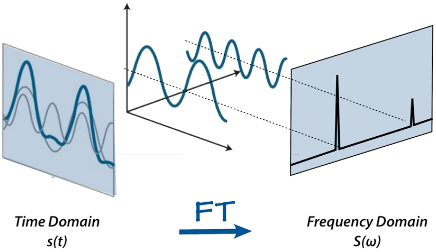
$$\mathcal{F}(x \star h) = \mathcal{F}(x) \cdot \mathcal{F}(h)$$

Proof:

The Fourier Transform .com

$$\mathscr{F}\{g(t)\} = G(f) = \int_{-\infty}^{\infty} g(t)e^{-i2\pi ft}\,dt$$

$$\mathscr{F}^{-1}\{G(f)\} = g(t) = \int_{-\infty}^{\infty} G(f)e^{i2\pi ft}\,df$$



Discrete Fourier transform

- Discrete Fourier transform (DFT), denoted as $\mathcal{F}(x)$ is a vector such that each element is given by

$$\tilde{x}_m = \sum_{j=0}^{N-1} x_j e^{-i\frac{2\pi}{n}jm}$$

  where $i = \sqrt{-1}$

- Inverse DFT

$$x_j = \frac{1}{N}\sum_{m=0}^{N-1} \tilde{x}_m e^{i\frac{2\pi}{N}jm}$$

- equivalently

$$\tilde{x} = Ax$$

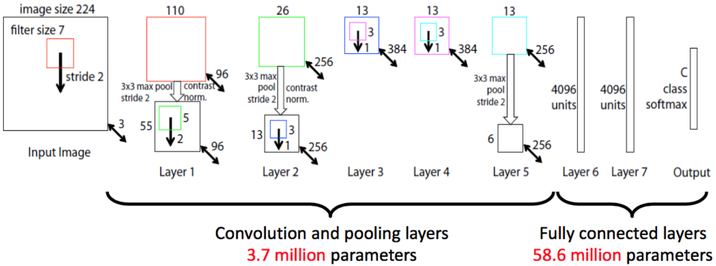
  where matrix A is N-by-N and $A_{mj} = e^{-\frac{2\pi i}{N}jm}$

Deep learning

Convolutional neural network (CNN)

Convolution + max pooling: highly structured processing pipeline

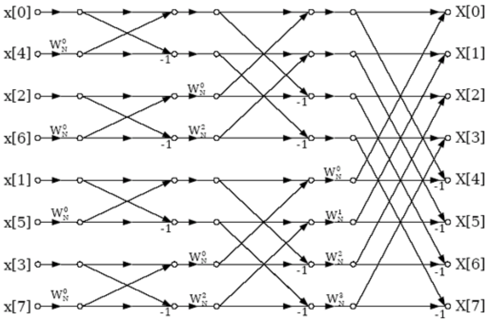
Fully connected layers: nonlinear classification



[Courtesy: Prof. Le Song at Georgia Tech, CSE.]

FFT (Fast Fourier Transform)

- Recall: DFT is equivalent to computing $\tilde{x} = Ax$
- Normally this is $\mathcal{O}(N^2)$, when the matrix has special form, however, it may be reduced. This is the idea of *fast Fourier Transform (FFT)*.
- Complexity of FFT is $\mathcal{O}(N \log N)$



butterfly

Derivation of FFT

- Assume N is even
- Let $e_n = x_{2n}$ represent the even-indexed samples
- Let $o_n = x_{2n+1}$ represent the odd-indexed samples
- One can show that e_n and o_n are zero outside the interval $0 \leq n \leq (N/2) - 1$

- One can show that

$$\tilde{x}_k = \frac{1}{2}\tilde{E}_k + \frac{1}{2}W_N^k\tilde{O}_k, \quad k = 0, 1, \dots, N-1$$

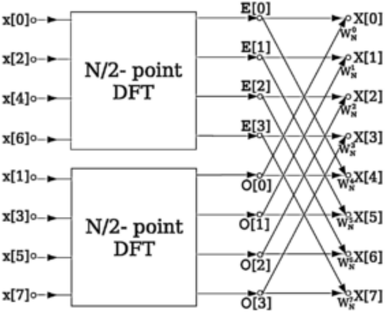
where $W_N = e^{-i\frac{2\pi}{N}}$
the two terms are DFT of the even- and the odd-indexed samples

$$\tilde{E}_k = 2\sum_{n=0}^{N/2-1} e_n W_{N/2}^{nk}, \quad \tilde{O}_k = 2\sum_{n=0}^{N/2-1} o_n W_{N/2}^{nk}$$

- Moreover, there is symmetry

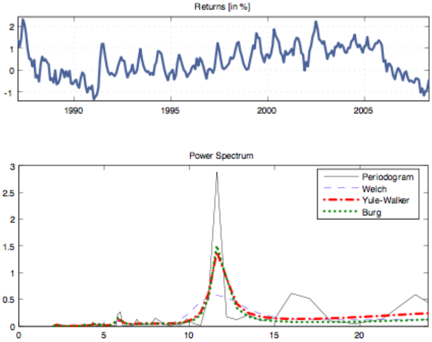
$$\tilde{E}_{k+N/2} = \tilde{E}_k, \quad \tilde{O}_{k+N/2} = \tilde{O}_k.$$

- *Length-N DFT of x_n can be computed as two DFTs of length $N/2$.*


x[0]
x[2]
x[4]
x[6]
x[1]
x[3]
x[5]
x[7]
N/2- point DFT
N/2- point DFT
E[0]
E[1]
E[2]
E[3]
O[0]
O[1]
O[2]
O[3]
X[0]
X[1]
X[2]
X[3]
X[4]
X[5]
X[6]
X[7]

Analysis of Financial Time-Series using Fourier method

- Case-Shiller home price index for the city of New-York
- January 1987 to May 2008 and the index is reported on a monthly basis



power spectral $|X(f)|^2$

strong seasonalities aect home prices and they have a frequency of recurrence of 12 months.

Summary

- Aspects of algorithms
  - Accuracy
  - Efficiency
  - Robustness and stability
- Analyzing algorithms
  - Bisection for finding root
  - Quicksort algorithm for sorting a sequence of numbers
  - Convolution and its quick implementation via FFT