

Algorithms



Computers

9

• Analog computers:

numbers are replaced by physical quantities,
the mathematical problem is simulated by
a physical one.

=> Example: slide rule

=> accuracy limited by physical measurements.

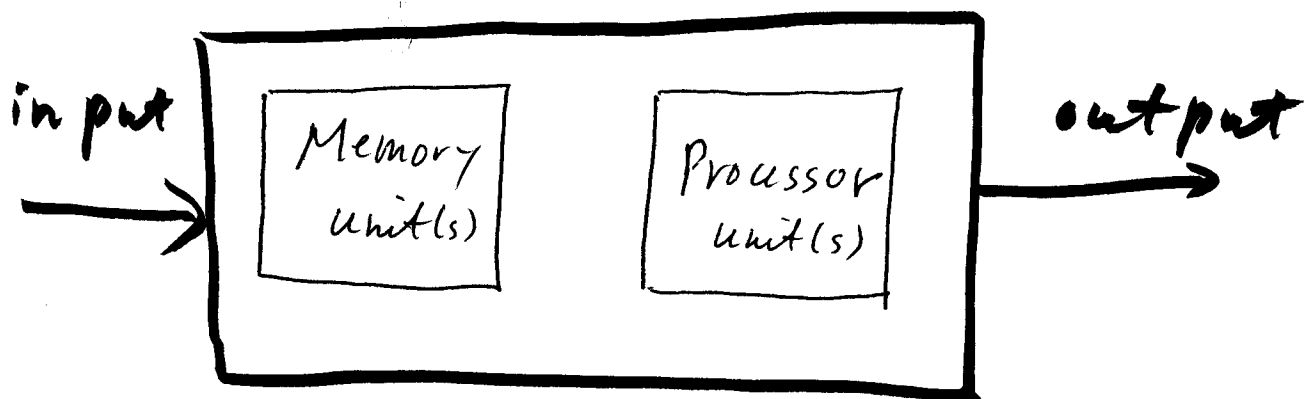
• Digital computers:

digits of numbers are represented by specific
physical quantities.

=> Example: use 21 and 23

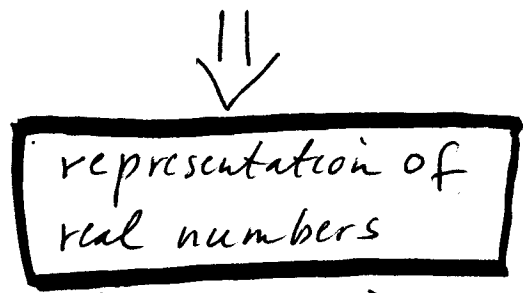
=> accuracy limited by the number of digits
rather than by the precision of physical measurements

• Principal layout of a computer:



Representation of Numbers

- Many physical entities are best described by real numbers.
- Simulation of physical processes related to such entities on a digital computer?

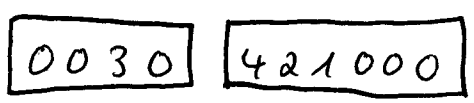


Fixed point rep.

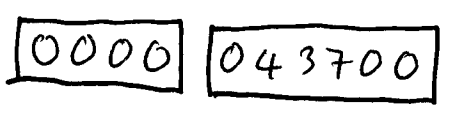
$$f = \pm \underbrace{d_1 \dots d_{n_1}}_{n_1 \text{ digits}} \cdot \underbrace{d_{n_1+1} \dots d_n}_{n_2 \text{ digits}}$$

Example: $n=10, n_1=4, n_2=6$

30.421



0.0437



Floating point rep.

$$f = \pm d_1 d_2 \dots d_t \times \beta^e$$

$$0 \leq d_i < \beta ; d_1 \neq 0$$

$$L \leq e \leq U, m \leq f \leq M$$

β : base

t : precision

$$\begin{cases} m = \beta^{L-1} \\ M = \beta^U (1 - \beta^{-t}) \end{cases}$$

$[L, U]$: exponent range

\Rightarrow all numbers representable by a given set of (β, t, L, U) are called machine numbers

Typical values: $(2, 56, -64, 64)$

Example: $\beta=2, t=3, L=0, U=2$

$$\begin{aligned} \pm 0.100_2 0 &\hat{=} \pm 1/2 \\ \pm 0.101_2 0 &\hat{=} \pm (\frac{4}{8} + \frac{1}{8}) = \pm \frac{5}{8} \\ \pm 0.110_2 0 &\hat{=} \pm (\frac{4}{8} + \frac{2}{8}) = \pm \frac{3}{4} \\ \pm 0.111_2 0 &\hat{=} \pm (\frac{4}{8} + \frac{2}{8} + \frac{1}{8}) = \pm \frac{7}{8} \end{aligned}$$

$$\begin{aligned} \pm 0.100_2 1 &\hat{=} 1 \\ \pm 0.101_2 1 &\hat{=} \frac{5}{4} \\ \pm 0.110_2 1 &\hat{=} \frac{3}{2} \\ \pm 0.111_2 1 &\hat{=} \frac{7}{4} \end{aligned}$$

$$\begin{aligned} \pm 0.100_2 2 &\hat{=} 2 \\ \pm 0.101_2 2 &\hat{=} \frac{5}{2} \\ \pm 0.110_2 2 &\hat{=} 3 \\ \pm 0.111_2 2 &\hat{=} \frac{7}{2} \end{aligned}$$

} machine numbers

• Due to historical memory limitations (~32 bit) computer arithmetic is sticking to floating point arithmetic.

• Major drawback => rounding errors !

Example 1: 16 digits precision

a = 0.109 632 969 473 378 2 10^0

b = 0.748 659 387 143 009 3 10^{-12}

Addition:
0.109 632 969 473 378 2
+ 0.000 000 000 000 748 659 387 143 009 3

lost due to rounding (12 digits!)

Example 2: 16 digits precision

$10^{20} + 17 - 10 + 130 - 10^{20} \longrightarrow 0$
 $10^{20} + 17 - 10^{20} - 10 + 130 \longrightarrow 120$
 $10^{20} - 10^{20} + 17 - 10 + 130 \longrightarrow 137$
 $10^{20} + 17 + 130 - 10^{20} - 10 \longrightarrow -10$

non-commutative operation on a computer !

• Addition of small and large numbers should be avoided.
But:

- we have to find a proper algorithm
- operations might be separated inside the code => hard to detect
- may depend on input data
- errors are hard to follow for large pieces of code.

Formal notion of an algorithm

12

$$X = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} \quad \text{input data}$$

$$Y = \begin{pmatrix} y_1 \\ \vdots \\ y_m \end{pmatrix} \quad \text{output data}$$

• Algorithm : $Y = \varphi(X)$

i.e. $y_i = \varphi_i(x_1, \dots, x_n) \quad i=1, \dots, m$

- At each stage of the calculation appears a set of numbers (input data + intermediate results (memory)) called the operand set:

$$X^{(i)} = \begin{pmatrix} x_1^{(i)} \\ \vdots \\ x_n^{(i)} \end{pmatrix}$$

• Elementary map : $X^{(i+1)} = \varphi^{(i)}(X^{(i)})$

\Rightarrow An algorithm may be decomposed into a sequence of elementary maps

$$\varphi = \varphi^{(r)} \circ \varphi^{(r-1)} \circ \dots \circ \varphi^{(0)}$$

Example: $a^2 - b^2 = (a+b)(a-b)$

Algorithm 1

$$n_1 := a \times a; n_2 := b \times b; Y := n_1 - n_2$$

Decomposition:

$$\varphi^{(0)}(a, b) = \begin{pmatrix} a^2 \\ b \end{pmatrix}, \quad \varphi^{(1)}(u, v) = \begin{pmatrix} u \\ v^2 \end{pmatrix}$$

$$\varphi^{(2)}(u, v) = u - v$$

Algorithm 2

$$n_1 := a + b; n_2 := a - b; Y := n_1 \times n_2$$

Decomposition:

$$\varphi^{(0)}(a, b) = \begin{pmatrix} a \\ b \\ a+b \end{pmatrix}; \quad \varphi^{(1)} = \begin{pmatrix} u \\ a-b \end{pmatrix}$$

$$\varphi^{(2)}(u, v) = u \cdot v$$

• Propagation of roundoff errors

- Algorithm:

$$x = x^{(0)} \rightarrow \varphi^{(1)}(x^{(0)}) = x^{(1)} \rightarrow \dots \rightarrow \varphi^{(r)}(x^{(r-1)}) = x^{(r)} = \varphi$$

- Remainder map:

$$\varphi^{(i)} = \varphi^{(i)} \circ \varphi^{(i-1)} \dots \circ \varphi^{(1)} \text{ with } \varphi^{(0)} = \varphi$$

- Jacobians:

$$D\varphi^{(i)}, D\varphi^{(i)}$$

$$D(f \circ g)(x) = Df(g(x)) \cdot Dg(x)$$

- With floating-point arithmetic, input and roundoff errors will perturb the (exact) intermediate results $x^{(i)}$:

$$\tilde{x}^{(i+1)} = fl(\varphi^{(i)}(\tilde{x}^{(i)}))$$

Absolute errors: $\Delta x^{(i+1)} = \tilde{x}^{(i+1)} - x^{(i+1)} = [fl(\varphi^{(i)}(\tilde{x}^{(i)})) - \varphi^{(i)}(\tilde{x}^{(i)})] + [\varphi^{(i)}(\tilde{x}^{(i)}) - \varphi^{(i)}(x^{(i)})]$

$$\approx D\varphi^{(i)}(x^{(i)}) \Delta x^{(i)}$$

$$fl(\varphi^{(i)}(\tilde{x}^{(i)})) - \varphi^{(i)}(\tilde{x}^{(i)}) \approx \underbrace{E_{i+1}}_{\text{Matrix!}} \cdot \varphi^{(i)}(x^{(i)}) := \alpha_{i+1}$$

(absolute roundoff error)

$$\Rightarrow \Delta x^{(i+1)} \approx \alpha_{i+1} + D\varphi^{(i)}(x^{(i)}) \cdot \Delta x^{(i)}$$

$$i \geq 0 \quad \Delta x^{(0)} = \Delta x$$

thus:

$$\Delta x^{(1)} \approx D\varphi^{(0)}(x) \Delta x + \alpha_1$$

$$\Delta x^{(2)} \approx D\varphi^{(1)}(x^{(1)}) [D\varphi^{(0)}(x) \cdot \Delta x + \alpha_1] + \alpha_2$$

$$\Delta y = \underbrace{D\varphi^{(r)} \dots D\varphi^{(0)}}_{D\varphi} \cdot \Delta x + \underbrace{D\varphi^{(r)} \dots D\varphi^{(1)}}_{D\varphi^{(r)}}$$

Example: $y = \varphi(a, b) = a^2 - b^2$

• Algorithm 1: $\begin{cases} x = x^{(0)} = \begin{pmatrix} a \\ b \end{pmatrix} \\ \text{elementary} \\ \text{maps} \end{cases} \begin{cases} x^{(1)} = \begin{pmatrix} a^2 \\ b \end{pmatrix} ; x^{(2)} = \begin{pmatrix} a^2 \\ b^2 \end{pmatrix} \\ x^{(3)} = y = a^2 - b^2 \end{cases}$

remainder maps $\begin{cases} \varphi^{(1)}(u, v) = u - v^2 ; \varphi^{(2)}(u, v) = u - v \\ D\varphi = (2a, -2b) ; D\varphi^{(1)}(x^{(1)}) = (1, -2b) \\ D\varphi^{(2)}(x^{(2)}) = (1, -1) \end{cases}$

$d_1 = \begin{pmatrix} \epsilon_1 a^2 \\ 0 \end{pmatrix} ; E_1 = \begin{pmatrix} \epsilon_1 & 0 \\ 0 & 0 \end{pmatrix} ; d_2 = \begin{pmatrix} 0 \\ \epsilon_2 b^2 \end{pmatrix} ; E_2 = \begin{pmatrix} 0 & 0 \\ 0 & \epsilon_2 \end{pmatrix}$

$d_3 = \epsilon_3 (a^2 - b^2)$

Error: $\Delta y \approx 2a\Delta a - 2b\Delta b + a^2\epsilon_1 - b^2\epsilon_2 + (a^2 - b^2)\epsilon_3$

• Algorithm 2: $\begin{cases} x = x^{(0)} = \begin{pmatrix} a \\ b \end{pmatrix} \\ \text{elementary} \\ \text{maps} \end{cases} \begin{cases} x^{(1)} = \begin{pmatrix} a+b \\ a-b \end{pmatrix} \\ x^{(2)} = y = a^2 - b^2 \end{cases}$

$d_1 = \begin{bmatrix} \epsilon_1 (a+b) \\ \epsilon_2 (a-b) \end{bmatrix} ; d_2 = \epsilon_3 (a^2 - b^2) ; E_1 = \begin{pmatrix} \epsilon_1 & 0 \\ 0 & \epsilon_2 \end{pmatrix}$

Error: $\Delta y \approx 2a\Delta a - 2b\Delta b + (a^2 - b^2)(\epsilon_1 + \epsilon_2 + \epsilon_3)$

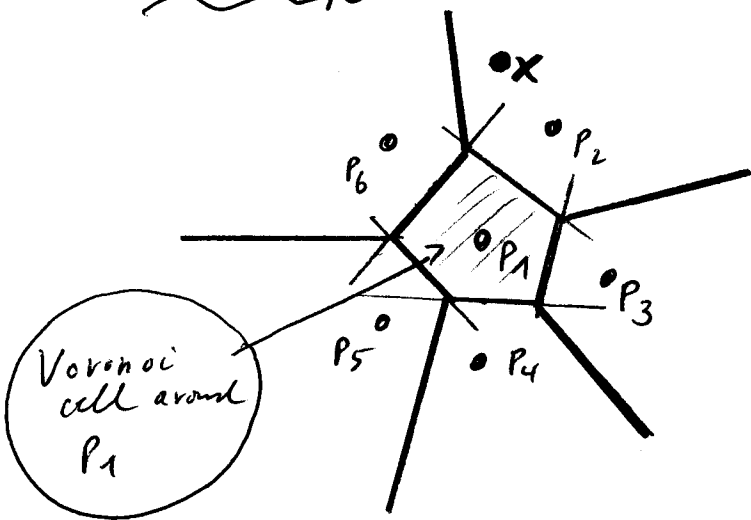
Assume: $\epsilon_i \approx \text{eps}$ (machine accuracy....)

total effect of rounding for Algo 2 is less than for Algo 1 if $\frac{1}{3} < \left|\frac{a}{b}\right|^2 < 3$

Listing

• A proper data structure may be the key to solve a given problem with the help of a computer!

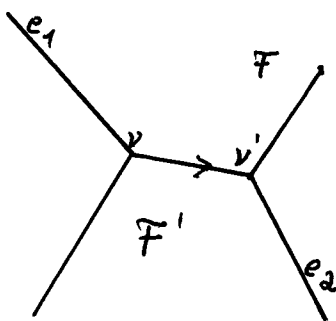
Example: Quantization (Analog-to-digital conversion)



Find the P_i closest to X .

Strategy 1: Determine distance between x and every P_i (inefficient).
 $O(N)$

Strategy 2: Carry out Voronoi program and determine Voronoi cell in which x is located.
 $O(\log N)$

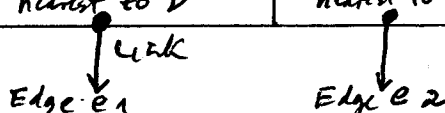


- Data structure (Voronoi diagram):

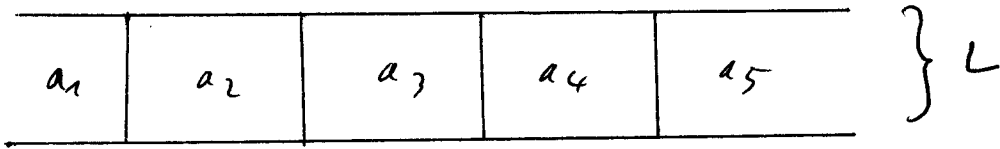
Edge

Vertex v (Origin)	Vertex v' (End point)
"left" cell F	"right" cell F'
edge of F nearest to v	edge of F' nearest to v'

=> Linked List



• Linear List

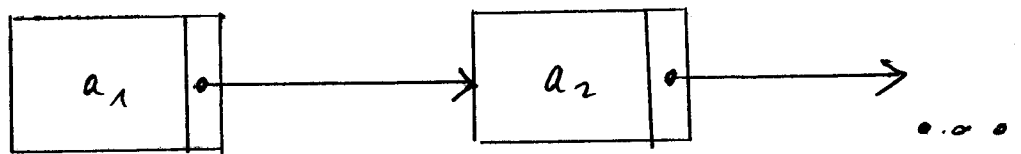


- Operations:
- $\text{Insert}(x, p, L)$: insert Element x at position p
 - $\text{Delete}(p, L)$: delete a_p from L
 - $\text{Search}(x, L)$: returns position p of element x
 - $\text{Get}(p, L)$: returns a_p
 -

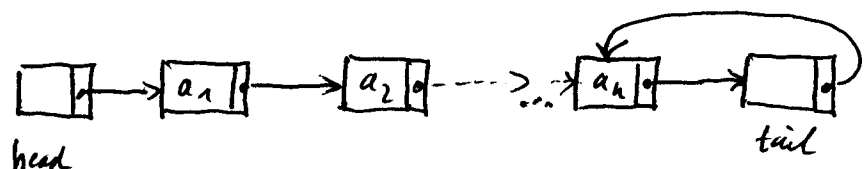
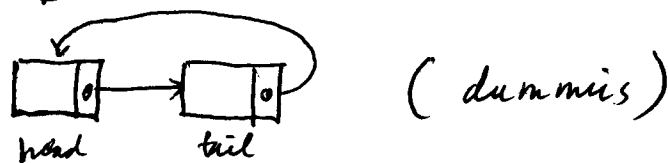
Storage: 1. sequential (see above)

2. linked

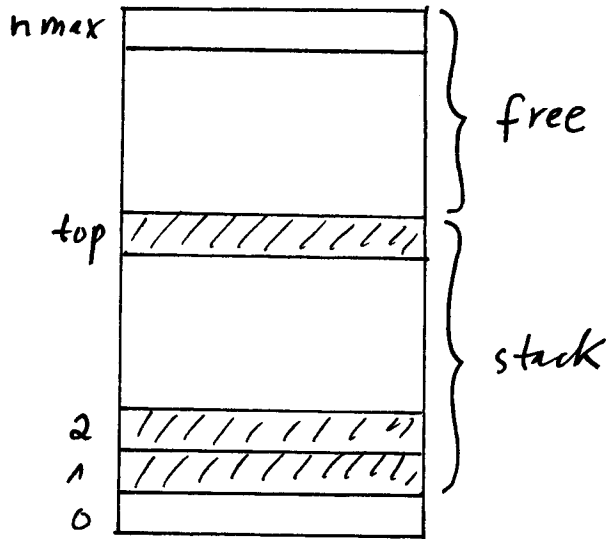
Elements are stored in "cells", which are connected by pointers



implementation:



stacks



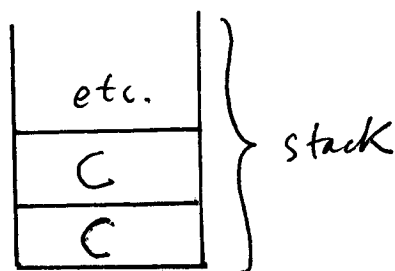
Operations:

- Pushhead (L, x) : insert x at the beginning of L
- Top(L) : return value of top element
- Push tail (L, x) : insert x at the end of L
- Pop head (L, x) : delete element at beginning of L
insert x
- Pop tail (L, x) : delete element at the end of L
insert x
- ⋮
- ⋮

Example:

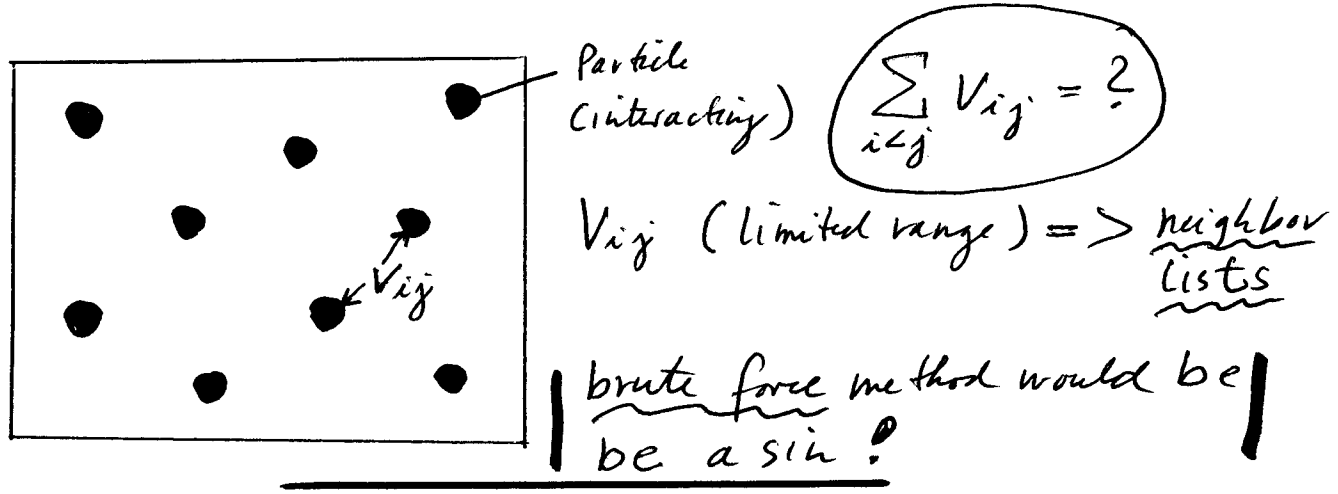
Check parentheses
 (()) ; ((()) ;

Read → (())



) : "deletes" top element of stack

o Special lists



Verlet neighbor list (2 arrays)

List : contains all neighbors for a given element

Point (i): points to the first neighbor of a given element i

\Rightarrow For a given element j, we have to scan list from Point(j) to Point(j+1)-1.

Linked cell technique

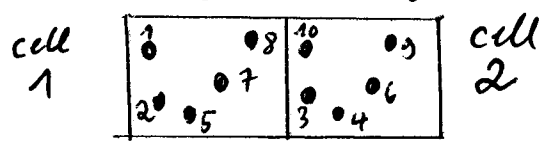
neighbors of cell 13

21	22	23	24	25
16	17	18	19	20
11	12	13	14	15
6	7	8	9	10
1	2	3	4	5

two arrays:

pos.	1	2	3	4	5	6	7	8	9	10
Head	8	10	—————							
List	0	1	0	3	2	4	5	7	6	9

- Head: contains "markers", i.e. an element of a given cell
- List: linked list array, contains next element in a given cell, 0 $\hat{=}$ acronym (end of listing)



\Rightarrow cell 2 (10, 9, 6, 4, 3); cell 1 (9, 7, 5, 2, 1)

Sorting

(20)

- The problem : Given a set of items a_1, \dots, a_n , and keys k_1, \dots, k_n (real numbers, " \leq "). Find permutation π such that:

$$\left| k_{\pi(1)} \leq k_{\pi(2)} \leq \dots \leq k_{\pi(n)} \right|$$

- Operations:
- a) comparisons between keys k and k' , i.e.
 $k = k'$, $k < k'$ or $k > k'$
 - b) elementary permutations between items a_i and a_j .

- Sorting by selection:

- Determine position j_1 with smallest key k_1 .
- Permute a_1 and a_{j_1} .
- Determine position j_2 with smallest key among a_2, \dots, a_n .
- Permute a_2 and a_{j_2} .

.....

Example:

j: 1, 2, 3, 4, 5, 6, 7
key: 15, 2, 43, 17, 4, 8, 47

Step 1: compare $a_1 \dots a_7$, permute a_1 and a_2
2, 15, 43, 17, 4, 8, 47

Step 2: compare $a_2 \dots a_7$, permute a_2 and a_5
2, 4, 43, 17, 15, 8, 47

Step 3: 2, 4, 8, 17, 15, 43, 47

Step 4: 2, 4, 8, 15, 17, 43, 47

• Sorting by insertion

- Pick $a_1 \dots a_n$ one by one, and insert them a_i into a sorted partial array $a_1 \dots a_{i-1}$.

- Insertion: compare k_i to k_{i-1}, k_{i-2} . If $k_j > k_i$, move k_j to the right of k_i .

Example: Card games.

Example:

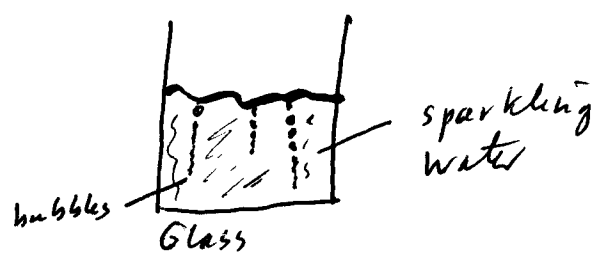
j: 1, 2, 3, 4, 5, 6, 7
key: 15, 2, 43, 17, 4, 8, 47

"Partial arrays":

2, 15 / 43, 17, 4, 8, 47 (step 2)
1 e.p. (elementary permutation)
2, 15, 17, 43 / 4, 8, 47 (step 4)
1 e.p.
2, 4, 15, 17, 43 / 8, 47 (step 5)
3 e.p.
2, 4, 8, 15, 17, 43 / 47 (step 6)
3 e.p. + (step 7)
0 e.p.

Bubble sort:

- compare each pair k_i, k_{i+1} while running through $a_1 \dots a_n$.
- if $k_i > k_{i+1}$, permute a_i and a_{i+1}
(biggest element to the right)
- Continue process until $a_1 \dots a_n$ are all sorted.



Example:

j: 1, 2, 3, 4, 5, 6, 7
key: 15, 2, 43, 17, 4, 8, 47
Step 1: 2, 15, 17, 4, 8, 43, 47

cont.

23
step 2: 2, 15, 4, 8, 17, 43, 47

step 3: 2, 4, 8, 15, 17, 43, 47

● Quicksort: (Divide and conquer)

- faster than the elementary methods described above.

- Algorithm (recursive):

Quicksort (F : sequence of keys)

If $F = \phi$ or $F = K_1$, F remains unchanged.

Divide: Pick pivot element K of F (f.e. the last one), and partition the rest of F into F_1 and F_2 , such that:

a) F_1 only contains elements $\leq K$

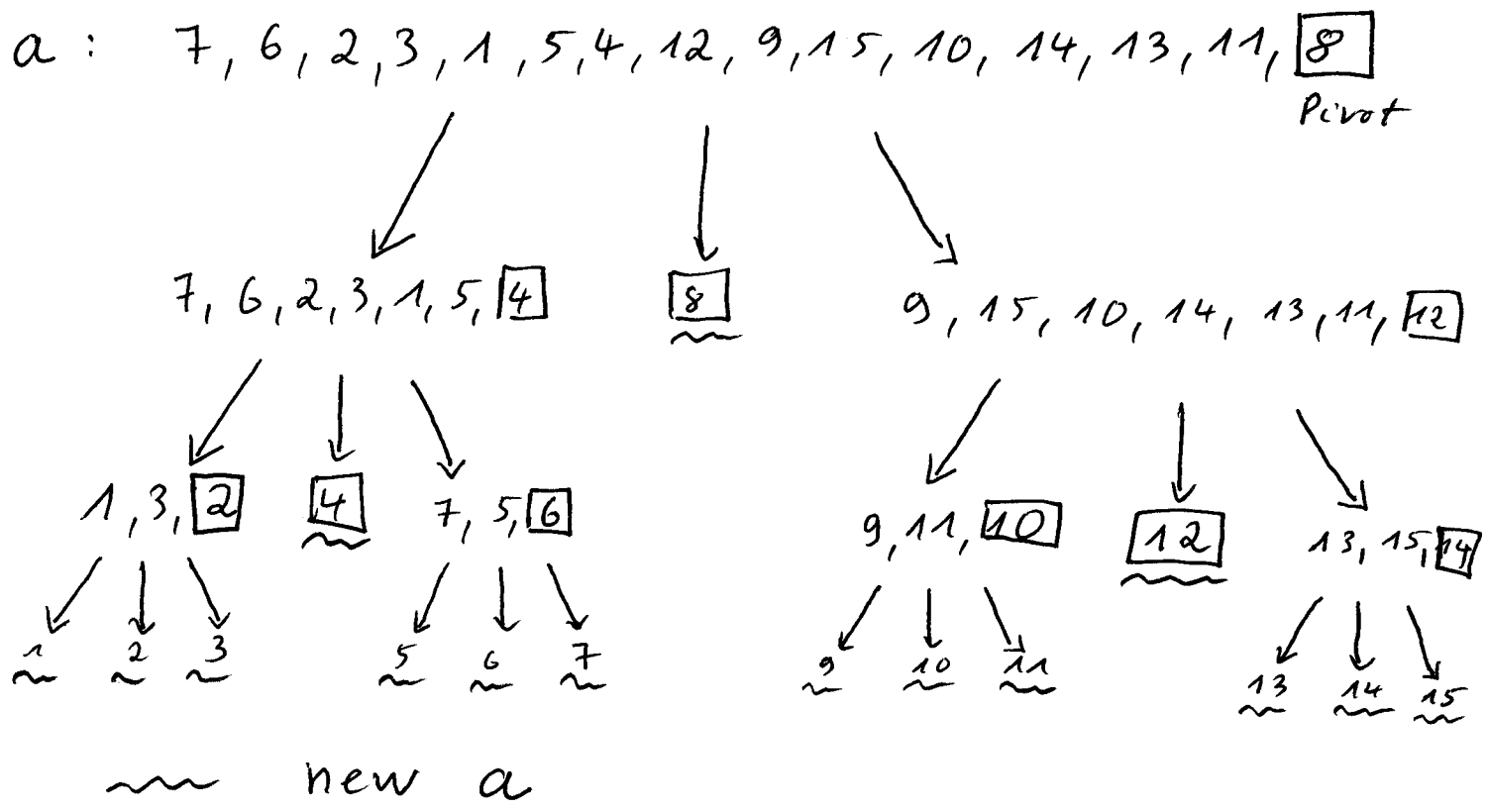
b) F_2 only contains elements $\geq K$

Conquer: Quicksort(F_1), Quicksort(F_2)

(After these calls, F_1 and F_2 are sorted)

Merge: Construct new F after concatenation of F_1, K, F_2 .

Example:



=> Study the various sorting algorithms using a card game (Skat, Poker, ...)

=> Other methods:

- Heapsort (see literature)
- Mergesort (see literature)

• • • • •

=> Can you beat Quicksort?
 (Has to be better than $O(\log N)$)
 \uparrow
size of array N

Searching

- The problem: Given a set of items a_1, \dots, a_N with keys k_1, \dots, k_N , find item(s) with "search key" K .

Assume: $k_1 \leq k_2 \leq \dots \leq k_N$

- Brute force method: run through your list \Rightarrow inefficient.

- Binary search: (List L)

- If $L = \emptyset$, the search is already over! Otherwise, pick element a_m at the "center" of L .

- If $K < a_m$, search a_1, \dots, a_{m-1} with the same method.

- If $K > a_m$, search a_{m+1}, \dots, a_N with the same method.

- Otherwise $K = a_m$, and we are done.

\Rightarrow Search can fail!

- More advanced methods:

GOOOOGLE

Random numbers

(28)

- Goal: Generate a set of random numbers within a given range (e.g. $(0 \dots 1)$).
 \Rightarrow uniform deviates: one number is as likely as all other numbers.

- Minimal standard generator:

Type: $I_{j+1} = a I_j + c \pmod{m}$

(generates a sequence of integers I_1, I_2, \dots)
each between 0 and $m-1$).

Park, Miller: $a = 7^5 = 16807$
 $m = 2^{31} - 1 = 2147483647$
 $c = 0$

- Variants:
- Schrage (implementation for 32 bit architectures)
 - Bays, Parkan (shuffling)

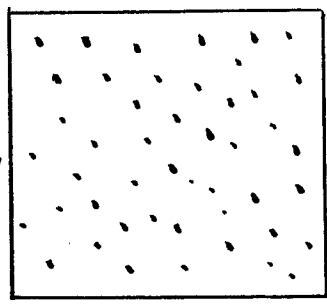
Period: $\approx 10^{18}$ (safe)

\Rightarrow Divide I_k by $(m-1)$, and you get random numbers in $(0 \dots 1)$.

• Tests:

Random number generator is supposed to generate random numbers within (0...1).

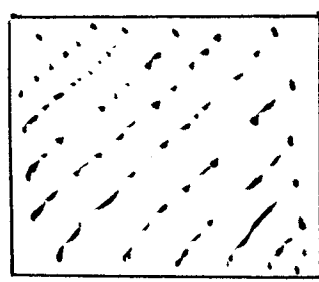
Good!



x

- Plot numbers successively in (x,y) pairs => should be randomly distributed in the unit square.

Forget it!



x

- Plot numbers successively in (x,y,z) triplets => should be randomly distributed in the unit cube.

etc.

• Transformations:

$$p(x) = \begin{cases} 1 & 0 < x < 1 \\ 0 & \text{otherwise} \end{cases} \Rightarrow \text{uniformly distributed in } (0...1) \text{ with } \int_{-\infty}^{+\infty} p(x) dx = 1$$

non-uniform distributions (deviates)

- Probability distribution: $p(y) dy$ where $y = \gamma(x)$

- Fundamental law of transformation of probabilities:

$$|p(y) dy| = |p(x) dx|$$

thus:
$$p(y) = p(x) \left| \frac{dx}{dy} \right|$$

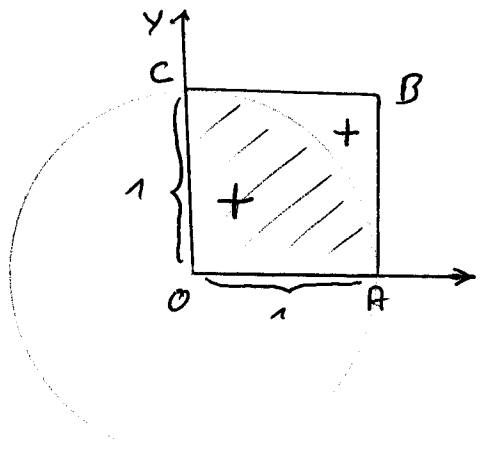
Example: $\gamma(x) = -\ln(x)$, $p(x)$ uniform in (0...1)

Exponential distribution

$$\Rightarrow p(y) dy = \left| \frac{dx}{dy} \right| dy = e^{-y} dy$$

Example:

π



- Hit and miss:

$$\pi \approx \frac{4 \times \text{area under curve } CA}{\text{area of square } OABC} = \frac{4 \mathcal{E}_{hit}}{\mathcal{E}_{shots}}$$

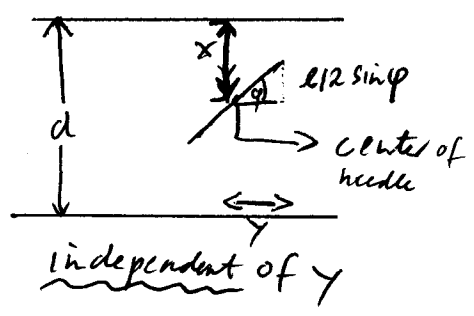
\mathcal{E}_{hit} : number of hits

\mathcal{E}_{shot} : number of shots

- Buffon's Needle:

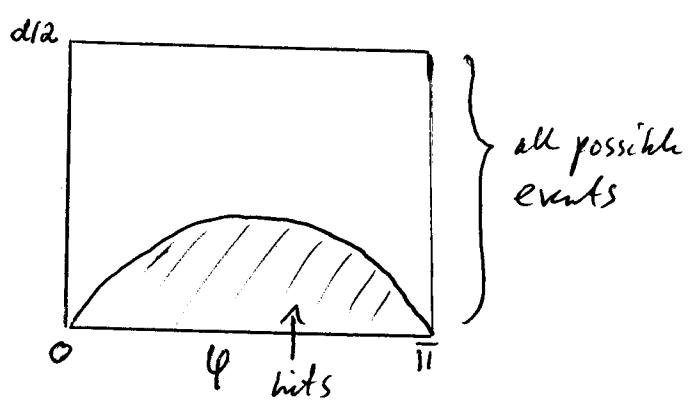
- draw a set of parallel lines (distance d) on the floor
- throw needle of length $l \leq d$
- count the number of hits (needle touches a line) and the number of throws

then: $\left| P(\text{hit}) = \frac{\# \text{ hits}}{\# \text{ throws}} = \frac{2l}{d\pi} \right|$



generally: $0 \leq x \leq d/2$
 $0 \leq \varphi \leq \pi$

hit: $x \leq l/2 \sin \varphi$



$$P(\text{hit}) = \frac{\int_0^{\pi} \frac{l}{2} \sin \varphi d\varphi}{d/2 \pi}$$

$$= \frac{l}{d\pi} [-\cos \varphi]_0^{\pi}$$

$$= \frac{2L}{d\pi}$$

Monte Carlo Integration

• Simple method for integrating functions $f(x)$.

• Procedure: • Pick N random points x_1, \dots, x_N from multidimensional volume V .

• Then:
$$\int f dV \approx V \langle f \rangle \pm \underbrace{V \sqrt{\frac{\langle f^2 \rangle - \langle f \rangle^2}{N}}}_{\text{error}}$$

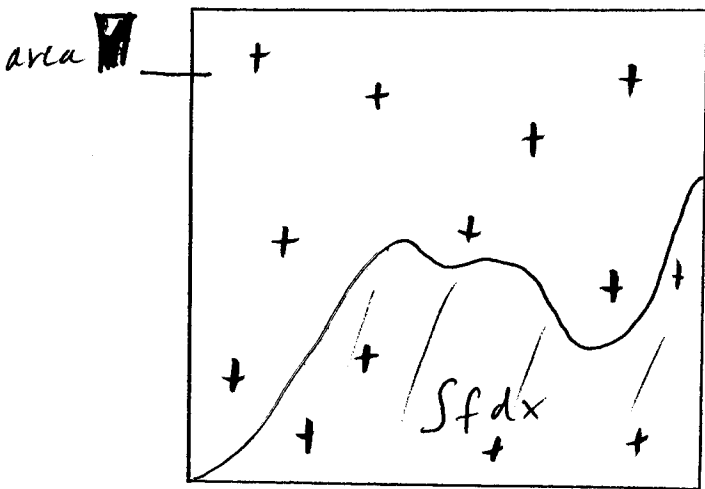
$$\langle f \rangle = \frac{1}{N} \sum_{i=1}^N f(x_i)$$

$$\langle f^2 \rangle = \frac{1}{N} \sum_{i=1}^N f^2(x_i)$$

• Problem: Integrate g over complex region W .

• Find V with $W \subset V$, which can easily be sampled.

• Define $f = \begin{cases} g & \text{in } W \\ 0 & \text{otherwise} \end{cases}$



$$\int f dx = A \cdot \frac{\# \text{ hits}}{\# \text{ shots}}$$

• Advanced procedures: see Numerical recipes !