F. Algorithms

V. Blobel
Inst. für Experimentalphysik, Hamburg

Sommersemester 2002

Robert Sedgewick, Algorithms, Addison-Wesley (various editions)
Sara Base, Computer Algorithms, Addison-Wesley

William H. Press, Saul A. Teukolsky, William T. Vetterling, Brian P. Flannery, Numerical Recipes in . . . (various progr. lang.), Cambridge University Press

1. Analysis of algorithms

Complex problems are divided into smaller problems, which hopefully can be solved by standard algorithms.

Important standard problems in data organisation

- sorting of items according to a key (ascending, descending),
- search in a list.

Algorithms usually apply to a set of objects of a certain size n.

The dependence $f(n)$ of the computer time or the number of operations on n for larger values of n (leading dependence) is often an essential property of an algorithm.

Sometimes important:

$A(n) =$ average behaviour $\qquad W(n) =$ worst case behaviour

Algorithm complexity

The complexity of an algorithm is analysed w.r.t.

- time (number of instructions?)
- space (amount of memory?)
- description (simple or many special cases?)

Required: (at least asymptotic) estimates of the time (and space) complexity in terms of the number n.

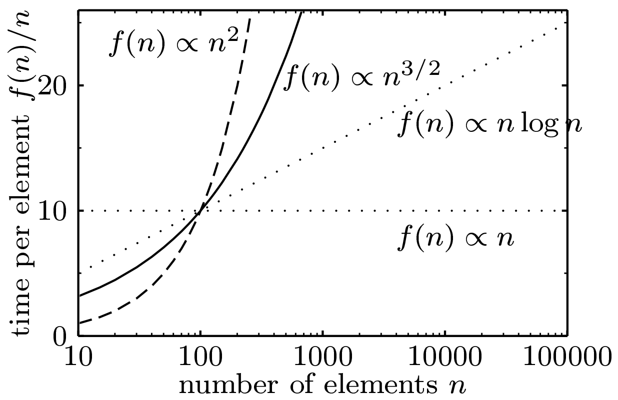
$$\text{Landau symbols:}\quad \begin{aligned} f &= O(g) \qquad \text{i.e.} \quad f(n) \le c \cdot g(n) \\ f &= o(g) \qquad \text{i.e.} \quad \lim_{n\to\infty} \frac{g(n)}{f(n)} = 0 \end{aligned}$$

$$\text{Example:}\quad f(n) = O(n \log n)$$

often written in the simplified form $f(n) = n \log n$.

Note: a statement ”the background is $O(10)$ events” makes no sense.

Typical leading dependences



function	sorting	searching
$f(n) = \text{const.}$		hash search
$f(n) = \text{const.} \cdot \log n$		binary search
$f(n) = \text{const.} \cdot n$		sequential search
$f(n) = \text{const.} \cdot n \log n$	quicksort, heapsort	
$f(n) = \text{const.} \cdot n^2$	quicksort with sorted input	
$f(n) = \text{const.} \cdot 2^n$		

Typical problems for algorithms

- compare the (100 000) words of a book with the 100 000 words of a dictionary (10^{10} comparisons?)
- find all hits on a certain wire in a drift chamber of 10 000 wires
- make straight line fit minimizing $\sum_i |r_i|$ with residuals r_i
- collect the energy from all cells in a calorimeter close to a given direction in φ and z (2 D search)
- in a MC detector simulation find the material at as given xyz-space point (3 D search)
- how many of the 10^6 MC events are close to an experimental event in a 10 D space (Q^2, x, y, rapidity, thrust, sphericity ...) (10 D search)

It is important to reduce the number of operations for n objects

$$O(n) \quad \rightarrow \quad O(\log n) \quad \rightarrow \quad O(1)$$

2. Data representation

Data representation is defined by the hardware. In general eight bits are combined into one byte.

There are two conventions used:

- systems that place the most significant byte at the lowest address in memory (*Big-Endian*), for example IBM 370 series, and
- systems that place the least significant byte at the lowest address (*Little-Endian*), for example Intel 8086 series.

Regardless of the byte ordering the least significant bit is generally referred as bit 0. The most significant bit is bit 7, bit 15 or bit 31, depending on wether the reference is to a byte, a half-word (two bytes), or a word (four bytes).

Integer representation

Integer quantities are usually represented as signed, two's complement numbers of two or four bytes, respectively.

one's complement of a number each bit is reversed (0 replaced by 1 and 1 replaced by 0).

two's complement of a number one's complement plus 1.

$$\begin{array}{rcrcr} 5_{10} & = & 0000\ 0101 & = & +5 \\ \text{ones-compl.} & = & 1111\ 1010 & & \\ \text{twos-compl.} & = & 1111\ 1011 & = & -5 \end{array}$$

The most significant bit of this number represents the sign.

Addition of a negative and a positive number:

$$\begin{array}{rcr} -5_{10} & = & 1111\ 1011 \\ +3_{10} & = & 0000\ 0011 \\ \hline -2_{10} & = & 1111\ 1110 \end{array}$$

Note: addition of 2 large positive numbers can have a negative number as a result.

Integer representation contnd.

Examples for two-byte integers

	bits		bits
$+1_{10}$	0000 0000 0000 0001	-1_{10}	1111 1111 1111 1111
$+2_{10}$	0000 0000 0000 0010	-2_{10}	1111 1111 1111 1110
$+3_{10}$	0000 0000 0000 0011	-3_{10}	1111 1111 1111 1101

An integer quantity can represent integer values between -32768 and $+32767$ (two bytes) and between $-2\,147\,483\,648$ and $+2\,147\,483\,647$ (four bytes).

Floating point representation

The IEEE floating-point representation (IEEE 754) has become a standard, which is particular widespread in the UNIX world.

IEEE 754: a single-precision floating-point number is a 32-bit word quantity, divided into three fields according to the table below.

31	30 23	22 0
sign	biased exponent	fractional part
1 bit	8 bits	23 bits

Single precision floating point representation

The value of floating-point number is

$$(-1)^{\text{sign}} \times (1 + \text{fraction}) \times 2^{(\text{biased exponent}-127)} .$$

The sign is 0 for a positive number, and 1 for a negative number.

The binary exponent may have values between -126 and $+127$. The *biased* exponent is the number's binary exponent plus a *bias* of 127; the biased exponent is always a positive integer between 1 and 254. The exponent values -127 and $+128$ are reserved for exception handling.

The *fraction* represent the fractional part of the mantissa. The representation is *normalized*: the integer part is *always* 1 and is not represented. Thus 23 bits are used for the fraction, but the precision corresponds to 24 bits.

Floating point range and precision

Single precision numbers (4 bytes) can have values in the range -3.7×10^{38} to $+3.7 \times 10^{38}$. The minimum magnitude of a floating-point number is 1.2×10^{-38}.

The relative precision is 0.6×10^{-7}.

Zero is represented by 0s in all fields.

Double-precision numbers (8 bytes) have 64 bits, with 11 bits for the biased exponent (bias of 1023) and 52 bits for the fractional part. Values in the range -1.7×10^{308} to $+1.7 \times 10^{308}$ can be represented and the minimum number is 2.3×10^{-308}.

The relative precision is 1.1×10^{-16}.

IEEE 754 Floating point error conditions

- Positive or negative overflow is represented by positive or negative infinity, printed *Inf* and *−Inf*.
- An indeterminant or impossible result like arccos(1.2) is represented by "Not a number", printed *NaN*.

Details of exception handling depend upon the system's implementation.

When infinities and NANs appear in computations, they behave in the expected way; for example any operation on a NaN produces another NaN. Comparisons of NaN with normal numbers are nasty (neither smaller nor smaller).

Characters, logical operations

Characters. Characters, i.e. the characters of the alphabet in lower and upper case and special characters like $ and & are usually stored in a (8 bit) byte. For the ASCII characters only 7 bits are necessary.

Logical operations. The basic logical operators operating on single bits are AND, OR and NOT. The unary operator NOT inverts the bit: NOT(0)=1 and NOT(1)=0. The definition of the operators AND and OR for single bits is shown below. Additional logical operators can be constructed form the basic operators. For example EOR (exclusive or) shown in the table too can be expressed by EOR(a,b) = OR(a,b)AND(NOT(AND(a,b))).

AND	0	1
0	0	0
1	0	1

OR	0	1
0	0	1
1	1	1

EOR	0	1
0	0	1
1	1	0

Arrays

Arrays with n dimensions are declared by $A(k_1, k_2, \ldots k_n)$, where the indices start either with 0 or 1, and increase in steps of 1. More general the index range may be i_j bis k_j, with declaration

$$A(i_1 : k_1,\, i_2 : k_2, \ldots i_n : k_n) \quad \text{with} \quad i_j \le \text{Index}_j \le k_j \quad \text{for} \quad j = 1, 2, \ldots n\ .$$

The total number of elements in the declaration of A is $M = \prod_{j=1}^{n}(k_j - i_j + 1)$.

The *lexical ordering* is used in languages like C and Pascal; the element $A_{j_1\, j_2 \ldots j_n}$ is store as the q-th element of the array with

$$q = 1 + \sum_{s=1}^{n-1}\left((j_s - i_s)\cdot \prod_{l=s+1}^{n}(k_l - i_l + 1)\right) + (j_n - i_n) \qquad \text{(row-wise)}\ .$$

Inverse lexical ordering is used in Fortran:

$$q = 1 + (j_1 - i_1) + \sum_{s=2}^{n}\left((j_s - i_s)\prod_{l=1}^{s-1}(k_l - i_l + 1)\right) \qquad \text{(column-wise)}\ .$$

3. Sorting – rearranging elements of a list

Basic algorithm in data processing with operations *compare* and *exchange* with properties

$$\text{from} \quad f(n) \propto n^2 \quad \rightarrow \quad f(n) \propto n \lg n$$

and with *in place*-algorithms and algorithms requiring certain extra space.

Selection Sort: the smallest element is searched for, and it is exchanged with the element in the first place; then the second smallest key is searched for and exchanged with the element in the second place, and so on.

$$A(n) = W(n) = (n-1) + (n-2) + \ldots + 2 + 1 = \frac{n(n-1)}{2} \approx \frac{n^2}{2}$$

Insertion and Shell sort

Insertion sort: assume that the first elements are already sorted. One step of the algorithm is inserting the next element x in its proper place, by moving each key in the sorted part by one place until a key smaller or equal to x is found.

$$W(n) = \frac{n(n-1)}{2} \propto n^2 \qquad A(n) \approx \frac{n^2}{2} \propto n^2$$

Efficient for nearly sorted lists.

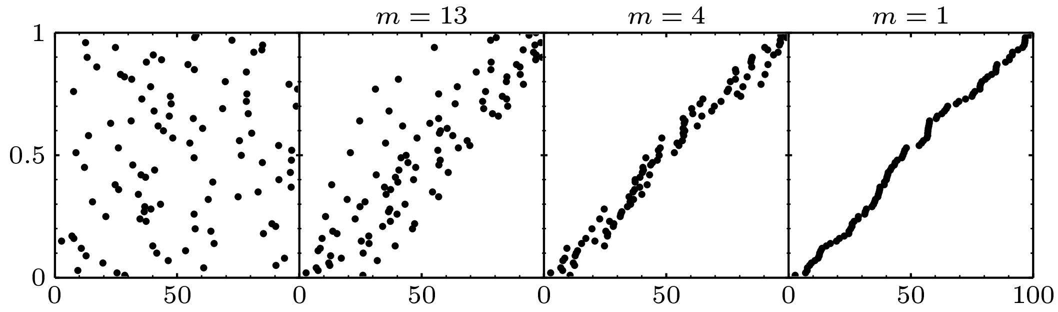
Shell sort: an increment m is used and defines m sublists, formed by elements with index increment m; each sublist is sorted by insertion sort. Increments are reduced, ending with $m = 1$. Analysis is difficult, but the algorithm easy to program and efficient:

$$A(n) \propto n^{3/2}$$

Good sequence of increments defined by recursion relation $m := 3 \cdot m + 1$ (up) and $m := (m-1)/3$ (down): $\ldots 40, 13, 4, 1$.

Example of Shell sort ...

... with increments $m = 13,\ 4,\ 1$



$n \ln n$ *algorithms – Quicksort*

The algorithm (by C.A.R. Hoare) is very fast for large n:

- select pivot: selection of one of the keys
- split: other keys are divided into two sets; one set is formed by all keys less than the pivot, and the second set is formed by all keys greater than or equal to the pivot.

The same procedure is then applied *recursively* to the subsets (or in an iteration).

The behaviour depends on the split: two subsets of about equal size are optimal. In the worst case Quicksort behaves poorly ($\propto n^2$); unfortunately sorted input is among the worst cases (extra strategies to avoid the worst-case behaviour).

Quicksort is not an in-place sort; a stack depending on n is used for the book keeping on the sublists, with a worst case $\propto n$.

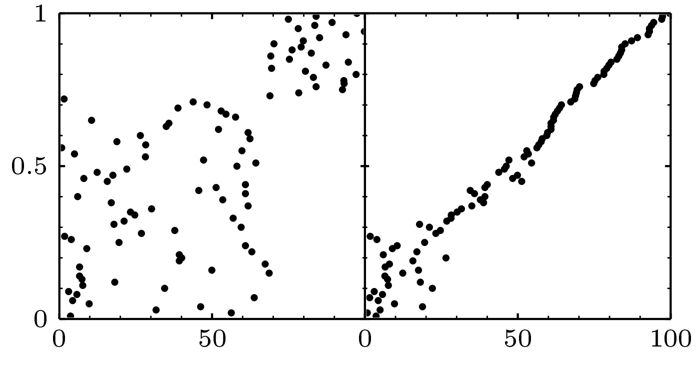
Simple version of Quicksort with recursion . . .

```
/* qsort: sort v[left]...v[right] in ascending order */
void qsort(int v[], int left, int right)
{
   int i, last;
   void swap(int v[], int i, int j);
   if(left >= right)           /* nothing to do if the vector */
      return;                  /* has less than two elements */
   swap(v, left, (left + right)/2);   /* move selected element */
   last = left;                       /* to v[0] */
   for (i = left+1; i <= right; i++)  /* split */
       if (v[i] < v[left])
           swap(v, ++last, i);
   swap(v, left, last);         /* get back selected element */
   qsort(v, left, last-1);
   qsort(v, last+1, right);
}

/* swap: exchange v[i] and v[j] */
void swap(int v[], int i, int j)
{
   int temp;
   temp = v[i];
   v[i] = v[j];
   v[j] = temp;
}
```


Quicksort example

The state after the first split, and before the final steps . . .

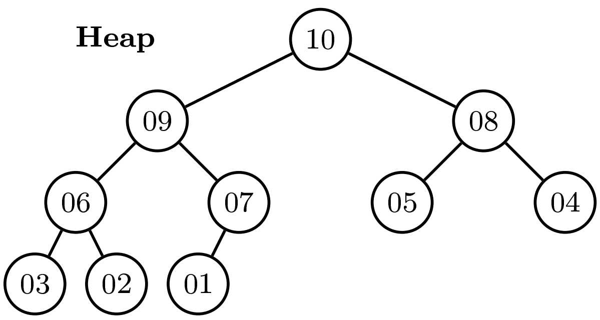


A data structure called heap

The data structure called *heap* is a special binary tree (all inner knots have two children, perhaps except one). To each of the n knots a number t_i for $i = 1, 2, \ldots n$ is assigned with relations

$$t_{\lfloor j/2 \rfloor} \geq t_j \qquad \text{for} \quad 1 \geq \lfloor j/2 \rfloor < j \leq n$$

($j/2$ is rounded to the next lower integer in the division $\lfloor j/2 \rfloor$). The element t_1 has thus the largest value.

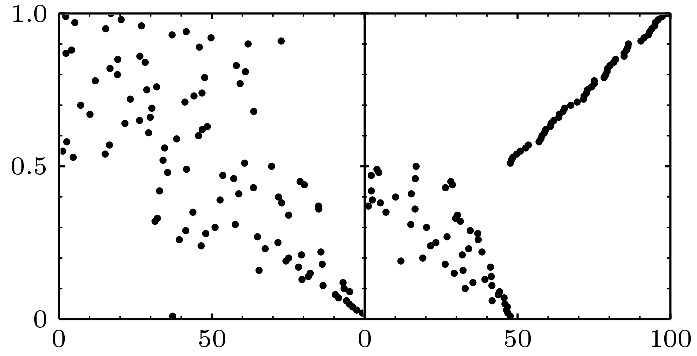


$n \ln n$ algorithms – Heapsort

Heapsort. The algorithm (by J. Williams) based on heaps is an in-place sort, and is usually the recommended algorithm for the general case. The worst case is only 20% slower than the average.

There are two stages in heapsort:

- first a heap data structure is constructed;
- then sorting is done from the high-end, removing always the largest key from the heap, and re-establishing the heap again.



Merging

Two sorted lists A and M with n and m elements are combined to a sorted list C with $n+m$ elements:

```
indexA=1;   indexB=1;   indexC=1;
while indexA =< n and indexB =< m do
   if A[indexA] < B[indexB]
      then   C[indexC] = A(indexA]; indexA = indexA + 1
      else   C[indexC] = B(indexA]; indexB = indexB + 1
   end if
   indexC = indexC + 1
end while

if indexB > m
   then  move A[indexA]...A[n]  to  C[indexC]...C[n+m]
   else  move B[indexB]...B[m]  to  C[indexC]...C[n+m]
end if
```

The algorithm needs in the worst case $n+m-1$ comparisons.

Merge sort

The merge algorithm can be used for sorting:

- divide list recursively into sublists of about equal length;
- the lowest level sublist are sorted
- all sublists are merged again.

The worst case is $W(n) = n \lg n - n$ but because of the extra space needed Merge sort is not an in-place sort.

External sort with 4 files

The number of keys that fit into high-speed memory at one time is m.

Task: sort n keys with $n \gg m$ into nondescending order, using external sort with four files, minimizing number of passes (with IO).

A *run* is a sequence of keys in nondecreasing order.

Phase I. Run construction and distribution: arrange the keys in nondecreasing order in two files

while there are more records in file T_0 **do**

1. read in m records (perhaps fewer the last time)
2. sort them using internal sort
3. add this run to file T_3, if the previous run went to file T_2; else to file T_2;

end;

Polyphase merge

Phase II. Repeatedly merge the runs until there is only one

$i_1 = 2$; $i_2 = 3$; indices of input files
$j_1 = 0$; $j_2 = 1$; indices of output files

while there is more than one run **do**

1. merge the first run in T_{i_1} with the first run in T_{i_2} and put resulting (longer) run in T_{j_1}.
2. merge the next run from T_{i_1} and T_{i_2} and put resulting (longer) run in T_{j_2}.
3. repeat steps 1 and 2, putting the output alternately in T_{j_1} and T_{j_2} until the end of files T_{i_1} and T_{i_2}
4. reset files (rewind). Add 2 modulo 4 to indices i_1, i_2 and j_1, j_2, to reverse the role of input and output files.

end;

The total number of passes is $N(n) = \lceil \lg n/m \rceil + 1$.

Median determination

The median is a robust estimate for the location of a distribution, defined by a finite sample of value x_i, $i = 1, \ldots n$

Straightforward method by sorting the x_i, using $k = (n+1)/2$:

$$\text{median} = \begin{cases} x_k & \text{if } n \text{ is odd,} \\ 1/2\,(x_k + x_{k+1}) & \text{if } n \text{ is even.} \end{cases}$$

- Is sorting necessary? Advantage: allows the selection of the m-th element for arbitrary index m.
- Is change of x-array allowed?
- Is extra space needed?

A statistical (randomized) method for the median

Algorithm:

- select "random" subset of $m = n^{3/4}$ elements and sort with e.g. Heapsort.
- select from sorted list the elements p_1 and p_2:

  $$p_1 \quad \text{with rank } m/2 - \sqrt{n} \qquad p_2 \quad \text{with rank } m/2 + \sqrt{n}$$

- count number of elements n_1 with $x < p_1$ and n_2 with $x > p_2$; copy all other elements.
- the copied list will contain the median if $n_1 < n/2$ and $n_2 < n/2$. If this is the case: sort copied list and extract median, otherwise repeat.

Required numnber of comparisions is $3/2n + o(n)$ with probability of $1 - O(n^{-1/4})$

Deterministic algorithms require at least $2n - o(n)$ comparisons

… but are still better even for $n = 10^4$.

Selecting the m-th largest element

Fast general method for selection (allowing rearrangement) is partitioning, exactly as done in Quicksort (see Press et al.):

- select a "random" partition element;
- go through array, forcing smaller elements to the left, larger elements to the right;
- ignore one subset and repeat with the subset which contains the desired m-th largest element, until element found.

No stack or extra space required (only one subset is used), but rearrangement is done. Median determination is a special case. Number of operations is $O(\log n)$.

4. Searching

Sequential search. Simple sequential search in a list for a given key requires a time proportional to the number of elements:

$$f(n) \propto n$$

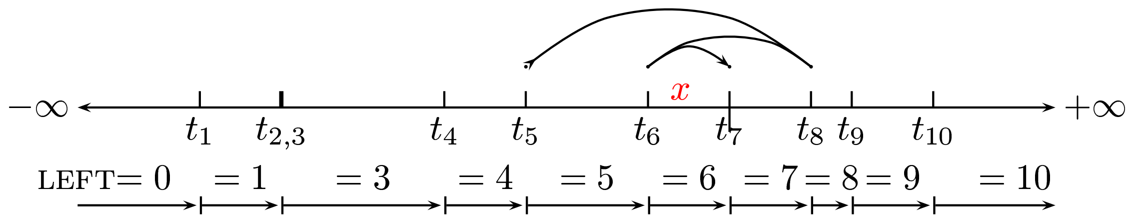
Binary search in sorted lists. Key x compared to the element of the table with index $(n+1)/2$. For $x < t_{(n+1)/2}$ further search is in this part of the table, otherwise in the second part of the table. This method is repeated for the reduced lists, dividing the length of the list each time by two

$$f(n) \propto 1 + \lceil \lg n \rceil$$

comparisons.

Binary search

Binary search for the interval with $t_\ell \leq x < t_{\ell+1}$



A binary tree is used to describe the sequence of comparisons made by a function searching in a sorted array.

Hash search

Simple adressing method: if the keys are integers in the range 0 to $(p-1)$ a table of length p can be used, where the key is directly taken as index. Large values of p require a lot of space.

Generalization: define hash function

$$\begin{array}{lrcl} & \text{hash(key)} & \rightarrow & \text{integer } \; 0\ldots(p-1) \\ \text{Example:} & \text{hash}(m) & = & m \mod p \end{array}$$

(with large value of p, preferentially a prime number) – the function value is used as index.

Collisions (where different keys are transformed to the same index) have to be resolved.

Often keys are long alphanumerical strings, which can be represented by an integer. Taking advantage of the properties of the modulo function the following code can be used to avoid very large integers:

```
j = ch(1); for i=2 to k do j = j*256+ch(i) mod p
```

Two data structures: the stack . . .

Pushdown stack – a simple restricted-access data structure with only two operations:

- push an item onto the stack (insert it at the beginning), and
- pop an item (remove it from the beginning).

A stack is like an "in" box: ... *work piles up in a stack, and is taken off the top.*

. . . ... and trees

A binary tree:

- every node is pointed to by only one other node (called parent);
- each node has left and right links;
- each node has an associated record with a key value.

A binary *search* tree:

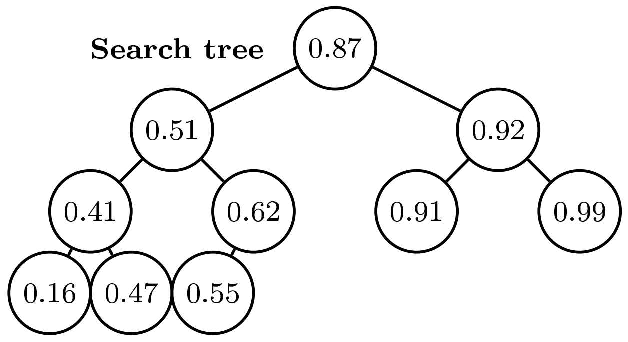
- a data structure of records connected with links, to be used for searching;
- all records with smaller keys are in the left subtree, and all records in the right subtree have larger (or equal) key values.

Construction of a binary search tree:

- leave keys untouched (no change of order etc.)
- build left/right link structure by operations *treeinitialize* and *treeinsert*

Binary search trees

. . . all records with smaller keys are in the left subtree, and all records in the right subtree have larger (or equal) key values.



Traversing a binary tree

How to traverse a tree, visiting systematically every node?
Recursive mathods are natural!

Method: "visit the root, then traverse the left subtree, then traverse the right subtree" (preorder traversal), or

"traverse the left subtree, the visit the root, then traverse the right subtree" (symmetric order), or

"visit the root, traverse the left subtree, traverse the right subtree" (preorder).

Preorder traversal

Recursive implementation of tree traversal is more natural than a stack-based implementaion, because trees are recursively defined structures . . .

recursive implementation:

```
procedure traverse (t : link);
  begin
  if t ≠ z then
    begin
    visit (t);
    traverse (t↑.l);
    traverse (t↑.r)
    end
  end;
```

non-recursive stack based implementation:

```
procedure traverse (t : link);
  begin
  push (t);
  repeat
    t := pop;
    visit (t);
    if t↑.r ≠ z then push (t↑.r);
    if t↑.l ≠ z then push (t↑.l);
  until stackempty
  end;
```

"Manual" recursion removal often leads to an efficient nonrecursive implementation.

Binary tree search and range searching

Binary tree search is a simple, efficient, dynamic serching method; it is one of the most fundamental algorithms in computer science.

Range searching =

finding all records in a data base that satisfy certain range restrictions on a specified set of attributes

The algorithm:

- Preprocessing: build the given points into a structure, based on links;
- range-searching: use the structure to return points falling into a given range.

One-dimensional range searching

Property: one-dimensional range searching can be done with

- $O(N \lg N)$ steps for the preprocessing,
- and $O(R + \lg N)$ for range searching, where R is the number of points actually falling in the range.

Generalization from $1D$ to $2D$ to kD trees is straightforward: cycle through the dimensions while going down the tree. Branching at each node corresponds to cutting the k-dimensional region of interest with a $(k-1)$-dimensional hyperplane.

Property of 2D tree:

- Construction of a 2D tree from N random points requires $2N \ln N$ comparisons, on average;
- Range searching with a 2D tree seems to use about $R + \log N$ steps to find R points in ranges in a region containing N points.

Recursive code for 1D tree

```
type interval = record xl,xr: integer end
procedure treerange (t : link); (int: interval);
  var bxl, bxr: boolean;
  begin
  if t ≠ z then
    begin
    bxl := t↑.key ≥ int.xl;
    bxr := t↑.key ≤ int.xr;
    if bxl then treerange(t↑.l,int);
    if bxl and bxr
      then t↑.key is within the range
    if bxr then treerange(t↑.r,int);
    end
  end;
```

5. Density estimation

…from a large sample of data x_i, $i = 1 \ldots n$.

Methods:

- Histograms. The standard method. Result is discontinuous at bin boundaries. Requires bin and limit definition before filling. Memory space requirement small.
- k-th nearest neighbor density estimation

  $$\hat{f}(x_i) \propto \frac{2k}{x_{i+k} - x_{i-k}}$$

  Efficient for the search for narrow peaks in a large range. Requires sorting of all elements.
- Rosenblatt estimator. Estimator is fraction of counts that lie in a window of a certain width. Counts within the window can be weighted e.g. with Gaussian of certain width.

  Generalization to many dimensions possible but requires some memory space (all $\boldsymbol{x}_i$) and a fast algorithm – range search.

  Can be used for problems, which usually are treated with artificial neural nets.

k-th nearest neighbor density estimator

Density estimate is counts per distance interval, with predetermined probability content $2k/N$, and the interval size $h(N)$ required to contain the probability is taken from the data:

$$\hat{f}(x_i) \propto \frac{k}{N \cdot h(N)} = \frac{k}{N \cdot (x_{i+k} - x_{i-k})}$$

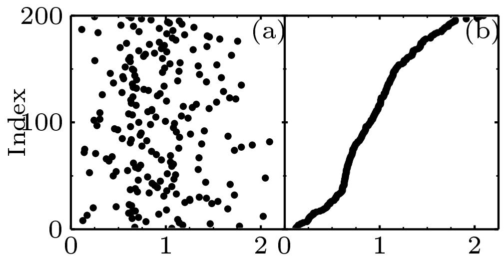
$$\text{Relative uncertainty} \quad \frac{\sigma\left[\hat{f}(x)\right]}{\hat{f}(x)} \propto \frac{1}{\sqrt{k(N)}} \quad \text{is constant}$$

$$\text{Expectation value} \quad E\left[\hat{f}(x)\right] = f(x) + \frac{1}{24}\left[\frac{k(N)}{N}\right]^2 \frac{f''(x)}{f^2(x)}$$

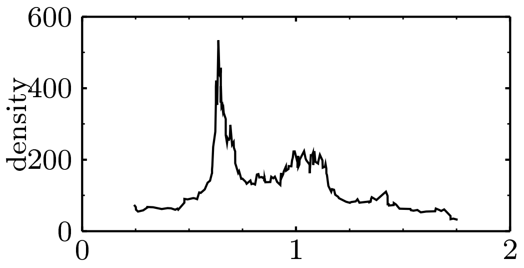
i.e. bias $\propto$ nonlinearity of the probability density function, approaching zero as $k(N)/N$ approaches zero.

k-nearest neighbor density estimator

(a) Unsorted and (b) sorted data sample of 200 values.



density $\propto \frac{k}{x_{i+k} - x_{i-k}}$



Rosenblatt estimator

Estimator is fraction of counts that lie in a window of width $2h$:

$$\hat{f}(x) = \frac{\hat{F}(x+h) - \hat{F}(x-h)}{2h} = \frac{N_x(h)}{2h}$$

where $\hat{F}(x)$ is the empirical cumulative distribution of the data points.

Expectation value $E\left[\hat{f}(x)\right] = \frac{F(x+h) - F(x-h)}{2h} = f(x) + \frac{h^2}{6} f''(x) + \ldots$

i.e. bias $\propto f''(x)$ approaching zero quadratically, as the window size h approaches zero.

Variance of the estimate: $V\left[\hat{f}(x)\right] = \frac{f(x)}{2hN} + \frac{h^4}{36} \left|f''(x)\right|^2 + \ldots$

…minimized for $h = CN^{1/5}$ with $C = \left[\frac{9f(x)}{2\left|f''(x)\right|^2}\right]^{1/5}$