Lecture 36
Sorting

- Straight insertion
- Shell's method
- Quicksort
- Heapsort
- Indexing
- Selecting M largest

Admittedly, the topic of today's lecture can be catalogued as "details". Yet some practical knowledge of sorting techniques is indispensable for any computational physicist.

Sorting consists of ordering data sets according to some criteria. One may also want to know other things like "median" value or "upper quartile"

So today we will discuss:

-Sorting, that is rearranging an array of numbers into numerical order.
-Rearranging an array while performing the corresponding rearrangement in one or more additional arrays so the correspondence between elements is kept.
-Given an array, prepare an index table for it, that is, a table of pointers telling which number array element comes first in numerical order, which second and so on.
-Select the Mth largest element from an array.

For the basic task of sorting N elements, the best algorithms require of the order of several times $N \log_2 N$ operations. The algorithm inventors try to reduce the number in front of that expression as much as possible.

Straight insertion

Straight insertion is quick an easy, but is an N^2 algorithm. It should be used only for small sets (say N<20).

The technique of it is the one card sorters use: pick the second card and put it in order with respect to the first, then pick out the third card and insert it into the sequence among the other first two and so on until the last card has been picked and inserted.

```
      SUBROUTINE piksrt(n,arr)
      INTEGER n
      REAL arr(n)
          Sorts an array arr(1:n) into ascending numerical order, by straight insertion. n is input;
          arr is replaced on output by its sorted rearrangement.
      INTEGER i,j
      REAL a
      do 12 j=2,n                        Pick out each element in turn.
          a=arr(j)
          do 11 i=j-1,1,-1               Look for the place to insert it.
              if(arr(i).le.a)goto 10
              arr(i+1)=arr(i)
          enddo 11
          i=0
10        arr(i+1)=a                     Insert it.
      enddo 12
      return
      END
```

What if you also want to rearrange an array brr at the same time as ou sort arr?
Simply move an element of brr when you move one in arr.

```
      SUBROUTINE piksr2(n,arr,brr)
      INTEGER n
      REAL arr(n),brr(n)
          Sorts an array arr(1:n) into ascending numerical order, by straight insertion, while making
          the corresponding rearrangement of the array brr(1:n).
      INTEGER i,j
      REAL a,b
      do 12 j=2,n                    Pick out each element in turn.
          a=arr(j)
          b=brr(j)
          do 11 i=j-1,1,-1           Look for the place to insert it.
              if(arr(i).le.a)goto 10
              arr(i+1)=arr(i)
              brr(i+1)=brr(i)
          enddo 11
          i=0
10        arr(i+1)=a                 Insert it.
          brr(i+1)=b
      enddo 12
      return
      END
```

Shell's method

Shell's method is a variant of straight insertion, but a very powerful one.

Say one wishes to sort 16 numbers $n_1,\ldots,n_{16}$. First sort, by straight insertion, each of the 8 groups of 2 numbers $(n_1,n_9),(n_2,n_{10}),\ldots,(n_8,n_{16})$. Next, sort the 4 groups of 4 numbers $(n_1,n_5,n_9,n_{13}),(n_2,n_6,n_{10},n_{14}),\ldots,(n_4,n_8,n_{12},n_{16})$. Next sort the two groups of 8 numbers $(n_1,n_3,n_5,n_7,n_9,n_{11},n_{13},n_{15}),(n_2,n_4,n_6,n_8,n_{10},n_{12},n_{14},n_{16})$. Finally, sort the 16 numbers.

What? If one ends up sorting the 16 numbers, what was the point of the other sortings. The idea is that the previous sorts will yield the 16 numbers in an almost sorted order, and the final sort will then rarely have to go beyond a few places to find the right spot for the element in question.

The spacings between the numbers sorted on each pass through the data (8,4,2,1 in the above example) are called the *increments*. A Shell sort is sometimes called a *diminishing increments sort*. There has been a lot of research into how to choose a good set of increments. It turns out that the above one is not the best. A better choice is the sequence $(3^k-1)/2,\ldots,40,13,4,1$ It can be shown that for such sequence the number of operations required is $N^{3/2}$

```
      SUBROUTINE shell(n,a)
      INTEGER n
      REAL a(n)
```
Sorts an array a(1:n) into ascending numerical order by Shell's method (diminishing increment sort). n is input; a is replaced on output by its sorted rearrangement.
```
      INTEGER i,j,inc
      REAL v
      inc=1                                  Determine the starting increment.
1     inc=3*inc+1
      if(inc.le.n)goto 1
2     continue                               Loop over the partial sorts.
          inc=inc/3
          do 11 i=inc+1,n                    Outer loop of straight insertion.
              v=a(i)
              j=i
3             if(a(j-inc).gt.v)then          Inner loop of straight insertion.
                  a(j)=a(j-inc)
                  j=j-inc
                  if(j.le.inc)goto 4
              goto 3
              endif
4             a(j)=v
          enddo 11
      if(inc.gt.1)goto 2
      return
      END
```

Quicksort

For large N this is the fastest sorting algorithm. It is a “partition exchange” sorting method: A “partition element” a is selected from the array. Then by pairwise exchange of elements, the original array is partitioned into two subarrays. At the end of the round of partitioning, the element a is in its final place in the array. All elements in the left subarray are less or equal than a while all the elements in the right subarray are larger or equal to aa. The process is then repeated on the left and right subarrays independently, and so on.

The partitioning process is carried out by selecting some element, say the leftmost, as the partitioning element a. Scan a pointer up the array until you find an element >a, and then scan another pointer down from the end of the array till you find an element <a. These two elements are clearly out of place for the final partitioned array, so exchange them. Continue this process until the pointers cross. This is the right place to insert a, and that round of partitioning is done. The question of the best strategy when an element is equal to the partitioning element is subtle: the answer is you should stop and do an exchange. R. Sedgwick, Commun. ACM 21, 847 (1978).

Quicksort requires an auxiliary array of storage of length 2 $\log_2$ N. When a subarray reaches some size M, it is faster to sort it by straight insertion. The Numerical Recipes routine implements that. The optimal setting for M is machine dependent, for most machines M=7 is the best.

Quicksort's average running time is fast, but its worst running time is awful: it becomes an N^2 method. And remarkably the worse times occur for arrays that are most ordered.

The quickness of Quicksort comes from the simplicity and efficiency of its inner loop.

```
      SUBROUTINE sort(n,arr)
      INTEGER n,M,NSTACK
      REAL arr(n)
      PARAMETER (M=7,NSTACK=50)
```
Sorts an array `arr(1:n)` into ascending numerical order using the Quicksort algorithm. `n` is input; `arr` is replaced on output by its sorted rearrangement.
Parameters: `M` is the size of subarrays sorted by straight insertion and `NSTACK` is the required auxiliary storage.
```
      INTEGER i,ir,j,jstack,k,l,istack(NSTACK)
      REAL a,temp
      jstack=0
      l=1
      ir=n
1     if(ir-l.lt.M)then                    Insertion sort when subarray small enough.
        do 12 j=l+1,ir
          a=arr(j)
          do 11 i=j-1,l,-1
            if(arr(i).le.a)goto 2
            arr(i+1)=arr(i)
          enddo 11
          i=l-1
2         arr(i+1)=a
        enddo 12
```

```
    if(jstack.eq.0)return
    ir=istack(jstack)
    l=istack(jstack-1)
    jstack=jstack-2
else
    k=(l+ir)/2
    temp=arr(k)
    arr(k)=arr(l+1)
    arr(l+1)=temp
    if(arr(l).gt.arr(ir))then
        temp=arr(l)
        arr(l)=arr(ir)
        arr(ir)=temp
    endif
    if(arr(l+1).gt.arr(ir))then
        temp=arr(l+1)
        arr(l+1)=arr(ir)
        arr(ir)=temp
    endif
    if(arr(l).gt.arr(l+1))then
        temp=arr(l)
        arr(l)=arr(l+1)
        arr(l+1)=temp
    endif
```

Pop stack and begin a new round of partitioning.

Choose median of left, center, and right elements as partitioning element a. Also rearrange so that a(l) $\leq$ a(l+1) $\leq$ a(ir).

```
      i=l+1                          Initialize pointers for partitioning.
      j=ir
      a=arr(l+1)                     Partitioning element.
3     continue                       Beginning of innermost loop.
        i=i+1                        Scan up to find element > a.
      if(arr(i).lt.a)goto 3
4     continue
        j=j-1                        Scan down to find element < a.
      if(arr(j).gt.a)goto 4
      if(j.lt.i)goto 5               Pointers crossed. Exit with partitioning complete.
      temp=arr(i)                    Exchange elements.
      arr(i)=arr(j)
      arr(j)=temp
      goto 3                         End of innermost loop.
5     arr(l+1)=arr(j)                Insert partitioning element.
      arr(j)=a
      jstack=jstack+2
        Push pointers to larger subarray on stack, process smaller subarray immediately.
      if(jstack.gt.NSTACK)pause 'NSTACK too small in sort'
      if(ir-i+1.ge.j-l)then
        istack(jstack)=ir
        istack(jstack-1)=i
        ir=j-1
      else
        istack(jstack)=j-1
        istack(jstack-1)=l
        l=i
      endif
    endif
    goto 1
    END
```

Heapsort

Not as fast as Quicksort, Heapsort is nevertheless popular. It is a true "in place" sort, requiring no auxiliary storage. It is an N $\log_2$ N process not only in average but also in the worst case order of input data. In fact the worst case is about 20% slower than average.

The full justification of the theory of Heapsort is complicated. We will just outline the general principles. D. Knuth, The art of computer programming vol 3 (1973).

A set of numbers is said to form a "heap" if it satisfies,

$$a_{j/2} \geq a_j \quad \text{for} \quad 1 \leq j/2 < j \leq N$$

Here j/2 means "integer divide", that is, the result is either an integer or rounded off to the next integer. This definition makes sense if you think of the numbers ordered in a tree with the top being node a_1.

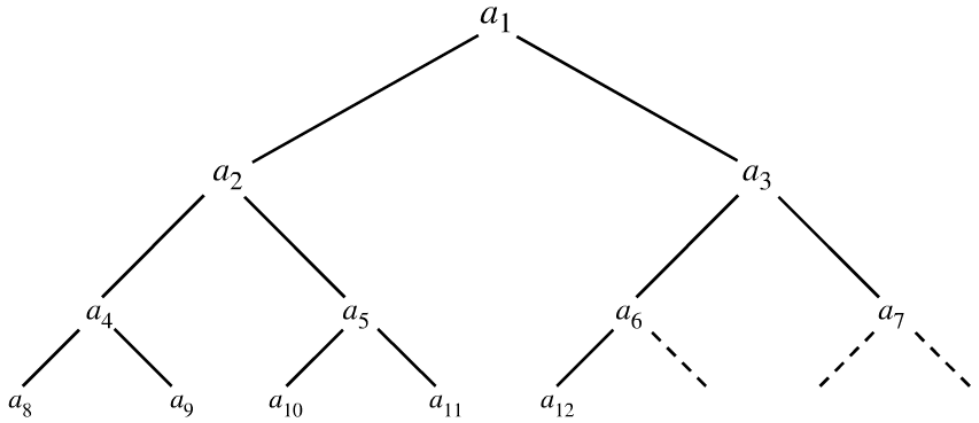

Figure 8.3.1. Ordering implied by a "heap," here of 12 elements. Elements connected by an upward path are sorted with respect to one another, but there is not necessarily any ordering among elements related only "laterally."

If you managed to rearrange your array into an order that forms a heap, then sorting is very easy: you pull off the “top of the heap”, which will be the largest element yet unsorted. Then you “promote” to the top of the heap its largest underling, and so on. An analogy is what happens in a large corporation when the chairman of the board retires. Evidently it is an $N \log_2 N$ process since each retiring chairman leads to $\log_2 N$ promotions of underlings.

So how do you arrange your array into a heap to begin with? The answer is again a “sift up” process like corporate promotion. Suppose that you start with N/2 employees without supervisors. A supervisor is hired to supervise two workers. If the supervisor is less capable than one of the workers, that workers is promoted in place of the supervisor. Then supervisors of supervisors are hired and so on. Each employee is brought in at the top of the tree, but then immediately sifted down, with more capable employees promoted until their proper corporate level has been reached.

```
      SUBROUTINE hpsort(n,ra)
      INTEGER n
      REAL ra(n)
```
Sorts an array `ra(1:n)` into ascending numerical order using the Heapsort algorithm. `n` is input; `ra` is replaced on output by its sorted rearrangement.
```
      INTEGER i,ir,j,l
      REAL rra
      if (n.lt.2) return
```
The index `l` will be decremented from its initial value down to 1 during the "hiring" (heap creation) phase. Once it reaches 1, the index `ir` will be decremented from its initial value down to 1 during the "retirement-and-promotion" (heap selection) phase.
```
      l=n/2+1
      ir=n
10    continue
        if(l.gt.1)then              Still in hiring phase.
          l=l-1
          rra=ra(l)
        else                        In retirement-and-promotion phase.
          rra=ra(ir)                Clear a space at end of array.
          ra(ir)=ra(1)              Retire the top of the heap into it.
          ir=ir-1                   Decrease the size of the corporation.
          if(ir.eq.1)then           Done with the last promotion.
            ra(1)=rra               The least competent worker of all!
            return
          endif
        endif
        i=l                         Whether in the hiring phase or promotion phase, we here
        j=l+l                           set up to sift down element rra to its proper level.
20      if(j.le.ir)then             "Do while j.le.ir:"
          if(j.lt.ir)then
            if(ra(j).lt.ra(j+1))j=j+1       Compare to the better underling.
          endif
          if(rra.lt.ra(j))then      Demote rra.
            ra(i)=ra(j)
            i=j
            j=j+j
          else                      This is rra's level. Set j to terminate the sift-down.
            j=ir+1
          endif
        goto 20
        endif
        ra(i)=rra                   Put rra into its slot.
      goto 10
      END
```

Selecting the Mth largest

Selection is related to sorting. The fastest methods for selection, unfortunately, rearrange the array for their own computational purposes, typically putting all smaller elements to the left of the kth, all larger to the right, and scrambling the order within each subset. When the array is large so making a copy of it is burdensome, or when the selection is a minor part of the total computation attempted, one can turn to selection algorithms without side effects, which leave the original array undisturbed, at the expense of some extra computational cost.

The most common use of selection is the statistical characterization of a set of data. One often wants to know the median element in an array, or the top or bottom quartile elements. When N is odd, the median is the kth element with k=(N+1)/2. When N is even, statistics books define the median as the arithmetic mean of the elements k=N/2 and k=N/2+1 (that is N/2 from the top and bottom). This is a bit irrelevant in practice, particularly if N is large, so you can stick with N/2 irrespectively.

The fastest general method for selection, allowing rearrangements, is partitioning, exactly as it was done in the Quicksort algorithm. Selecting a random partition element, one marches through the array, forcing smaller elements to the left, larger elements to the right.

```
      FUNCTION select(k,n,arr)
      INTEGER k,n
      REAL select,arr(n)
```
Returns the kth smallest value in the array `arr(1:n)`. The input array will be rearranged to have this value in location `arr(k)`, with all smaller elements moved to `arr(1:k-1)` (in arbitrary order) and all larger elements in `arr[k+1..n]` (also in arbitrary order).
```
      INTEGER i,ir,j,l,mid
      REAL a,temp
      l=1
      ir=n
1     if(ir-l.le.1)then                 Active partition contains 1 or 2 elements.
        if(ir-l.eq.1)then               Active partition contains 2 elements.
          if(arr(ir).lt.arr(l))then
            temp=arr(l)
            arr(l)=arr(ir)
            arr(ir)=temp
          endif
        endif
        select=arr(k)
        return
      else
        mid=(l+ir)/2                    Choose median of left, center, and right elements as par-
        temp=arr(mid)                     titioning element a. Also rearrange so that arr(l) ≤
        arr(mid)=arr(l+1)                 arr(l+1), arr(ir) ≥ arr(l+1).
        arr(l+1)=temp
```

```
        if(arr(l).gt.arr(ir))then
            temp=arr(l)
            arr(l)=arr(ir)
            arr(ir)=temp
        endif
        if(arr(l+1).gt.arr(ir))then
            temp=arr(l+1)
            arr(l+1)=arr(ir)
            arr(ir)=temp
        endif
        if(arr(l).gt.arr(l+1))then
            temp=arr(l)
            arr(l)=arr(l+1)
            arr(l+1)=temp
        endif
        i=l+1                       Initialize pointers for partitioning.
        j=ir
        a=arr(l+1)                  Partitioning element.
3       continue                    Beginning of innermost loop.
            i=i+1                   Scan up to find element > a.
        if(arr(i).lt.a)goto 3
4       continue
            j=j-1                   Scan down to find element < a.
        if(arr(j).gt.a)goto 4
        if(j.lt.i)goto 5            Pointers crossed. Exit with partitioning complete.
        temp=arr(i)                 Exchange elements.
         arr(i)=arr(j)
         arr(j)=temp
         goto 3                         End of innermost loop.
5        arr(l+1)=arr(j)                Insert partitioning element.
         arr(j)=a
         if(j.ge.k)ir=j-1               Keep active the partition that contains the kth element.
         if(j.le.k)l=i
    endif
    goto 1
    END
```

In-place, nondestructive selection is conceptually simple, but it requires a lot of bookkeeping, and as a consequence is slower. The general idea is to pick some number M of elements at random, sort them, and then make a pass through the array counting how many elements fall in each of the M+1 intervals defined by these elements. The kth largest will fall in one such interval (call it the "live" interval) and then determining which of the new, finer, M+1 intervals all presently live elements fall into. And so on, until the kth element is finally localized within a single array of size M, at which point direct selection is possible.

How shall we pick M? The number of rounds $\log_M N = \log_2 N / \log_2 M$, will be smaller if M is larger; but the work to locate each element among M+1 subintervals will be larger, scaling as $\log_2 M$ for bisection. Each round requires looking at all N elements, if only to find those that are still alive, while the bisections are dominated by the N that occur in the first round. Minimizing $O(N \log_M N) + O(N \log_2 M)$ yields the result,

$$M \sim 2^{\sqrt{\log_2 N}}$$

The square root of the log is such a slowly varying function that one can pick a safe number, say 64 and ignore the variation with N.

```
      REAL selip,arr(n),BIG
      PARAMETER (M=64,BIG=1.E30)
```
Returns the kth smallest value in the array `arr(1:n)`. The input array is not altered.
```
C     USES shell
      INTEGER i,j,jl,jm,ju,kk,mm,nlo,nxtmm,isel(M+2)
      REAL ahi,alo,sum,sel(M+2)
      if(k.lt.1.or.k.gt.n.or.n.le.0) pause 'bad input to selip'
      kk=k
      ahi=BIG
      alo=-BIG
1     continue                              Main iteration loop, until desired element is isolated.
        mm=0
        nlo=0
        sum=0.
        nxtmm=M+1
        do 11 i=1,n                         Make a pass through the whole array.
          if(arr(i).ge.alo.and.arr(i).le.ahi)then     Consider only elements in the cur-
            mm=mm+1                                       rent brackets.
            if(arr(i).eq.alo) nlo=nlo+1               In case of ties for low bracket.
            if(mm.le.M)then                 Statistical procedure for selecting m in-range elements
              sel(mm)=arr(i)                      with equal probability, even without knowing in
            else if(mm.eq.nxtmm)then          advance how many there are!
              nxtmm=mm+mm/M
              sel(1+mod(i+mm+kk,M))=arr(i)          The mod function provides a some-
            endif                                         what random number.
            sum=sum+arr(i)
```

```
            endif
          enddo 11
          if(kk.le.nlo)then                Desired element is tied for lower bound; return it.
            selip=alo
            return
          else if(mm.le.M)then             All in-range elements were kept. So return answer by
            call shell(mm,sel)                 direct method.
            selip=sel(kk)
            return
          endif                            Augment selected set by mean value (fixes degenera-
          sel(M+1)=sum/mm                        cies), and sort it.
          call shell(M+1,sel)
          sel(M+2)=ahi
          do 12 j=1,M+2                    Zero the count array.
            isel(j)=0
          enddo 12
          do 13 i=1,n                      Make another pass through the whole array.
            if(arr(i).ge.alo.and.arr(i).le.ahi)then      For each in-range element..
              jl=0
              ju=M+2
2             if(ju-jl.gt.1)then           ...find its position among the select by bisection...
                jm=(ju+jl)/2
                if(arr(i).ge.sel(jm))then
                  jl=jm
                else
                  ju=jm
                endif
              goto 2
              endif
              isel(ju)=isel(ju)+1          ...and increment the counter.
            endif
          enddo 13
          j=1                              Now we can narrow the bounds to just one bin, that
3         if(kk.gt.isel(j))then                  is, by a factor of order m.
            alo=sel(j)
            kk=kk-isel(j)
            j=j+1
          goto 3
          endif
          ahi=sel(j)
        goto 1
        END
```

Summary

- Straight insertion: reordering cards.
- Shell's method: insertion by groups.
- Quicksort operates by partition.
- Heapsort: org table of a company.
- Selection requires bookkeeping.