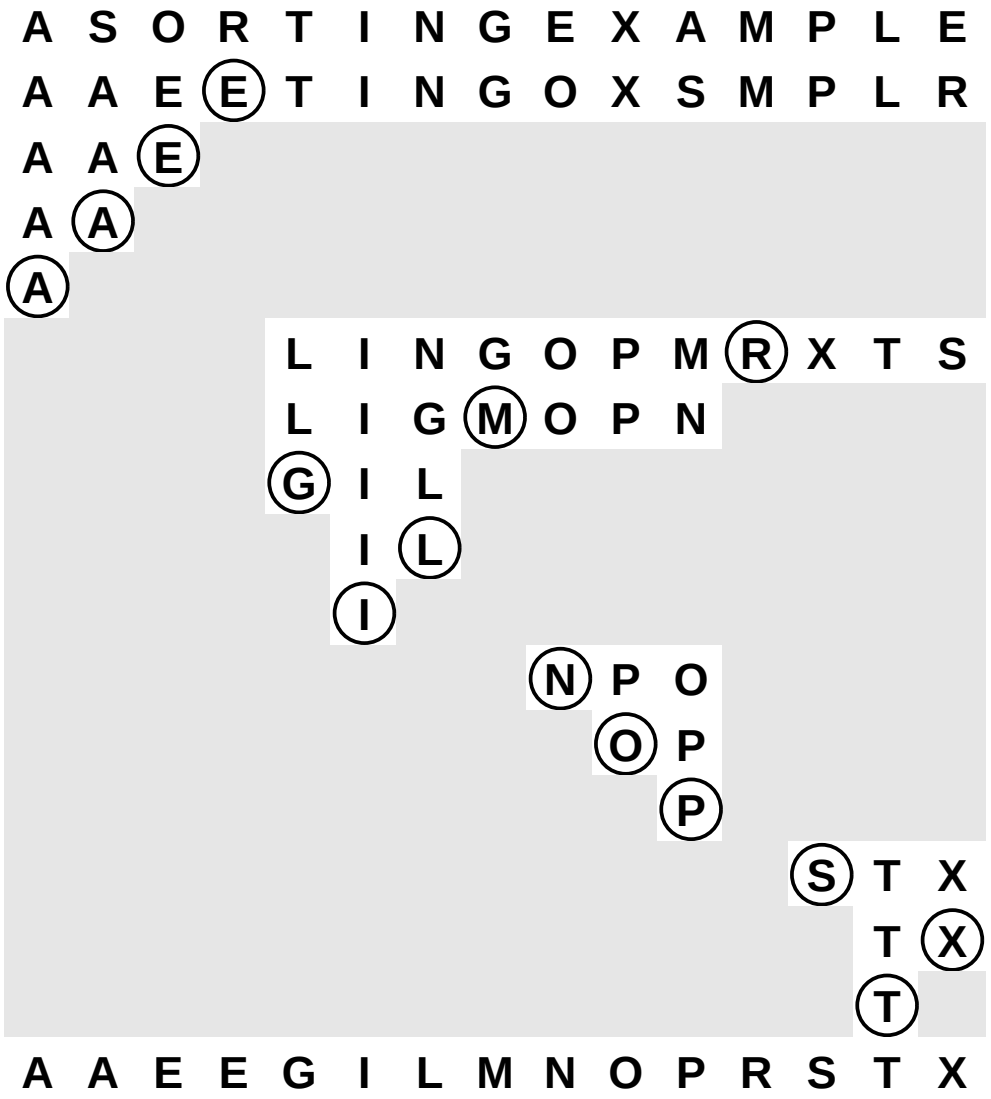
COS 226 Lecture 3: Quicksort

To sort an array, first divide it so that

* some element a[i] is in its final position
* no larger element left of i
* no smaller element right of i

Then sort the left and right parts recursively



Partitioning

To partition an array, pick a partitioning element

* scan from right for smaller element
* scan from left for larger element
* exchange
* repeat until pointers cross

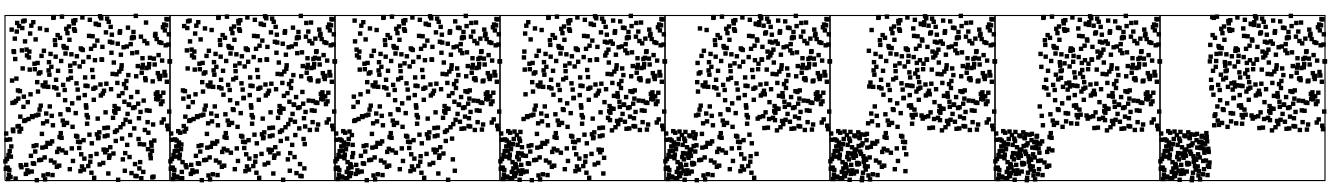
A S O R T I N G E X A M P L (E)

A S
A M P L
A A S M P L E

O
E X
A A E O X S M P L E

R
E R T I N G

A A E (E) T I N G O X S M P L R



Partitioning implementation

v: partitioning element

i: left-to-right pointer

j: right-to-left pointer

```
int partition(itemType a[], int l, int r)
  {
    int i, j; itemType v;
    v = a[r]; i = l-1; j = r;
    for (;;)
      {
        while (less(a[++i], v)) ;
        while (less(v, a[--j]))
          if (j == l) break;
        if (i >= j) break;
        exch(a[i], a[j]);
      }
    exch(a[i], a[r]);
    return i;
  }
```

Issues

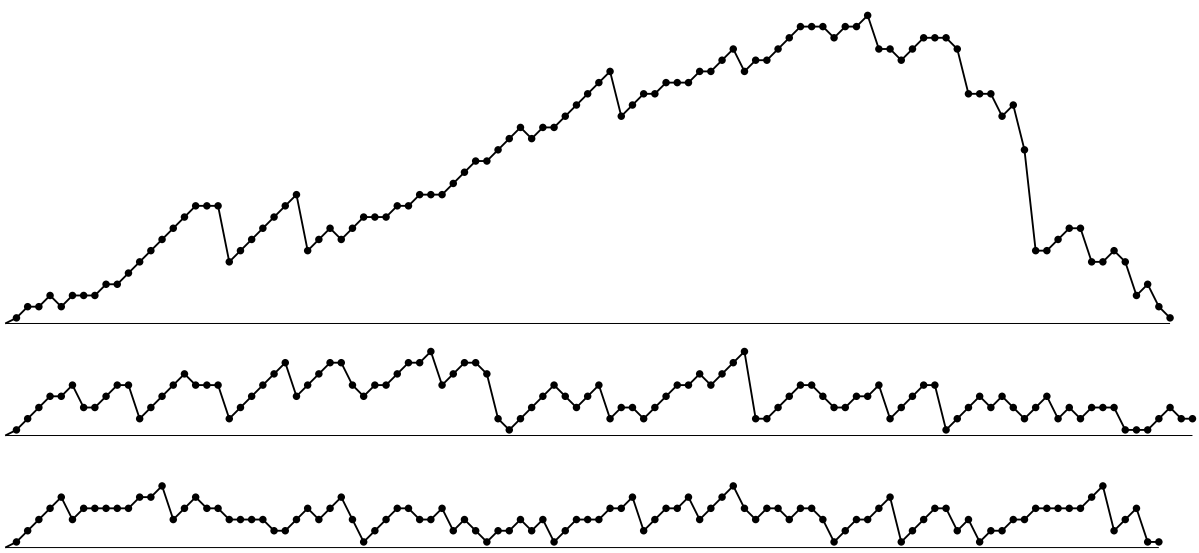
* stop pointers on keys equal to v?
* sentinels or explicit tests for array bounds?
* details of pointer crossing

Quicksort implementation

```
quicksort(int a[], int l, int r)
   {
      int i;
      if (r > l)
         {
            i = partition(l, r);
            quicksort(a, l, i-1);
            quicksort(a, i+1, r);
         }
   }
```

Issues

* overhead for recursion?
* running time depends on input
* worst-case time cost
  (quadratic, a problem)
* worst-case space cost
  (linear, a serious problem)



Nonrecursive Quicksort

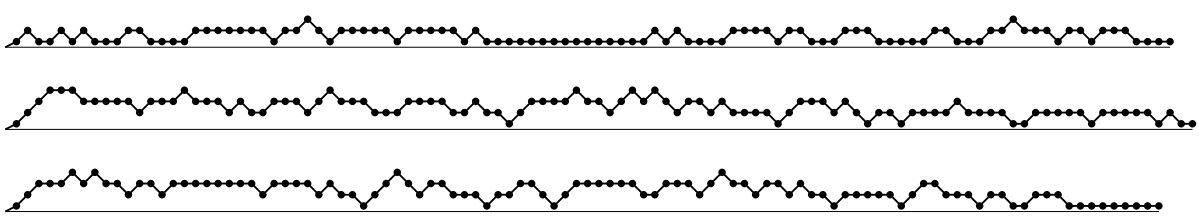
Use explicit stack instead of recursive calls

Sort smaller of two subfiles first

```
#define push2(A, B)  push(A); push(B);
void quicksort(itemType a[], int l, int r)
   {
      int i;
      stackinit(); push2(l, r);
      while (!stackempty())
         {
            r = pop(); l = pop();
            if (r <= l) continue;
            i = partition(a, l, r);
            if (i-l > r-i)
               { push2(l, i-1); push2(i+1, r); }
            else
               { push2(i+1, r); push2(l, i-1); }
         }
```

Guaranteed less than (lg N) extra space cost

Still quadratic worst-case time cost



Analysis

Total running time is sum of

cost*frequency

for all the basic operations

Cost depends on machine

Frequency depends on algorithm, input

For Quicksort

A -- number of partitioning stages

B -- number of exchanges

C -- number of comparisons

Cost on a typical machine

35A + 11B + 4C

Number of comparisons in the worst case

$N + (N-1) + (N-2) + \ldots = N(N-1)/2$

Average case analysis

Assume input randomly ordered

* each element equally likely to be partitioning element
* subfiles randomly ordered (if partitioning is "blind")

Average number of comparisions satisfies

$$C(N) = N+1 + (C(1) + C(N-1))/N + (C(2) + C(N-2))/N + \ldots + (C(N-1) + C(1))/N$$

$$C(N) = N+1 + 2(C(1) + C(2) + \ldots + C(N-1))/N$$

$$NC(N) = N(N+1) + 2(C(1) + C(2) + \ldots + C(N-1))$$

$$NC(N) - (N-1)C(N-1) = 2N + 2C(N-1)$$

$$NC(N) = (N+1)C(N-1) + 2N$$

$$C(N)/(N+1) = C(N-1)/N + 2/(N+1)$$

$$= 2(1 + 1/2 + 1/3 + \ldots 1/(N+1))$$

$$= 2 \ln N + (\text{small error term})$$

THM: Quicksort uses about $2N \ln N = (1.38\ldots)N \lg$ comparisons (and 1/6 as many exchanges) on average.

Randomized algorithm

* pick random partitioning element
* analysis then valid for any input

Inner loop

Use profiler

* find instructions executed most often
* verify analysis
* streamline program by iterating

```
quicksort(int a[], int l, int r)
   <132659>{
   int v, i, j, t;
   if (<132659>r > l)
      {
         <66329>v = a[r];
         <66329>i = l-1; <66329>j = r;
         for (<66329>;<327102>;<327102>)
            {
               while (<1033228>a[++i] < v) <639797>;
               while (<1077847>a[--j] > v) <684416>;
               if (<393431>i >= j) <66329>break;
          <327102>t = a[i]; a[i] = a[j]; a[j] = t;
            }
         <66329>t = a[i]; a[i] = a[r]; a[r] = t;
         <66329>quicksort(a, l, i-1);
         <66329>quicksort(a, i+1, r);
      }
   <132659>}
```

Another partitioning method

Seeking an easier-to-understand method
(detailed justification omitted)

```
quicksort(int a[], int l, int r)
<133395>{
int v, i, k, t;
if (<133395>r <= l) return;

<66697>v = a[l]; <66697>k = l;
for (<66697>i=l+1; <1976624>i<=r; <1909927>i++)
   if (<1909927>a[i] < v)
   { <934565>t = a[i]; a[i] = a[++k]; a[k] = t; }
   <66697>t = a[k]; a[k] = a[l]; a[l] = t;
   <66697>quicksort(a, l, k-1);
   <66697>quicksort(a, k+1, r);
   }

<133395>}
```

Not much simpler,
three times as many exchanges

Improvements to Quicksort

Median-of-three

* partitioning element closer to center
* estimate median with median of sample
* number of comparisons close to N lg N
  with large enough sample
* FEWER LARGE FILES
* slightly more exchanges, more overhead

Insertion sort small subfiles

* even Quicksort has too much overhead
  for files of a few elements
* use insertion sort for tiny files
* (can wait until the end)

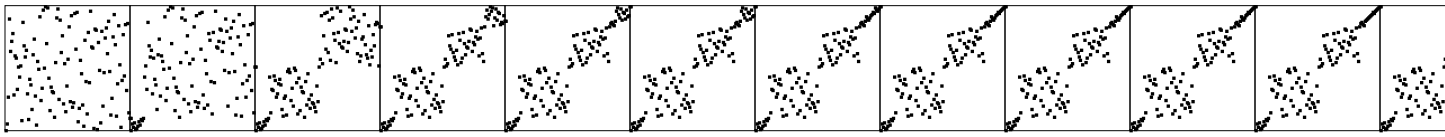
Analysis gives best value of parameters

* median of 3 elements
* cut to insertion sort for < 10 elements

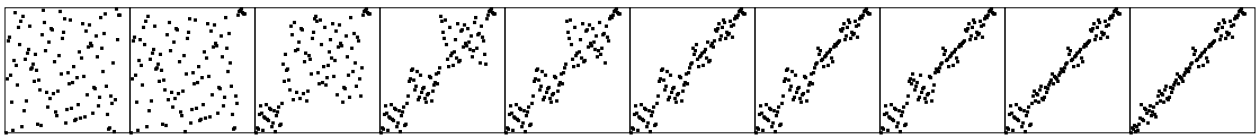
Improvements to Quicksort

Standard

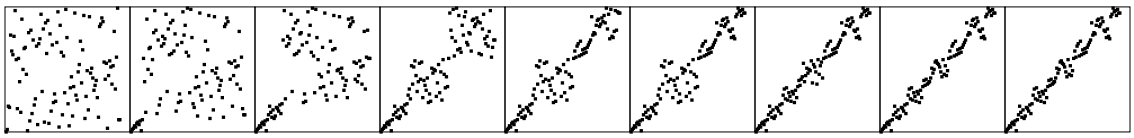
* might have bad partition (1st one in ex.)
* might have good partition (2nd one in ex.)



Cutoff for small subfiles



Median-of-three



Three-way partitioning

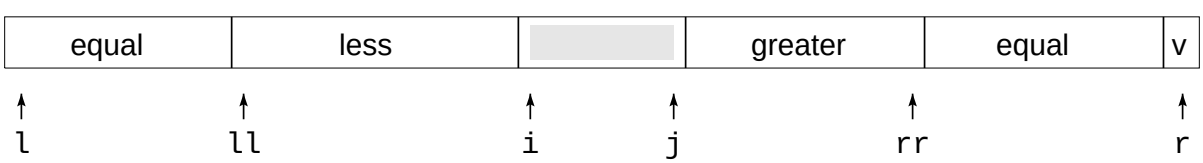
Equal keys can affect performance

Suppose all keys are the same
 * plain quicksort takes N lg N comparisons (!)
 * change partitioning to take N comparisons
 * naive method might use N^2 comparisons (!!)

Suppose there are two distinct key values
 * reduces to above case for one subfile
 * better to complete sort with one partition
        stop right pointer on left value
        stop left pointer of right value
              exchange

Large file, small number of key values
 * reduces to above cases

Partition into three parts
 * elements between i and j equal to v
 * no larger element left of i
 * no smaller element right of j



Three-way partitioning implementation

```
void exch(itemType a[], int i, int j)
   { itemType t = a[i]; a[i] = a[j]; a[j] = t; }

void quicksort(itemType a[], int l, int r)
{
   int i, j, k, ll, rr; itemType v;
   if (r <= l) return;

   v = a[r]; i = l-1; j = r; ll = l; rr = r;
   for (;;)
      {
         while (a[++i] < v) ;
         while (v < a[--j]) if (j == l) break;
         if (i >= j) break;
         exch(a, i, j);
         if (a[i] == v) exch(a, ll++, i);
         if (a[j] == v) exch(a, rr--, j);
      }
   exch(a, i, r); j = i-1; i = i+1;
   for (k = l  ; k<ll; k++, j--) exch(a, k, j);
   for (k = r-1; k>rr; k--, i++) exch(a, k, i);
   quicksort(a, l, j);
   quicksort(a, i, r);

}
```

Sorting strings

Use three-way partitioning on key characters

actinian	**coenobite**	**actinian**
jeffrey	**conelrad**	**bracteal**
coenobite	**actinian**	**coenobite**
conelrad	**bracteal**	**conelrad**
secureness	**secureness**	**cumin**
cumin	**dilatedly**	**chariness**
chariness	**inkblot**	**centesimal**
bracteal	**jeffrey**	**cankerous**
displease	**displease**	**circumflex**
millwright	**millwright**	**millwright**
repertoire	**repertoire**	**repertoire**
dourness	**dourness**	**dourness**
centesimal	**southeast**	**southeast**
fondler	**fondler**	**fondler**
interval	**interval**	**interval**
reversionary	**reversionary**	**reversionary**
dilatedly	**cumin**	**secureness**
inkblot	**chariness**	**dilatedly**
southeast	**centesimal**	**inkblot**
cankerous	**cankerous**	**jeffrey**
circumflex	**circumflex**	**displease**

Not many disadvantanges

(to be continued: this is a radix sort)