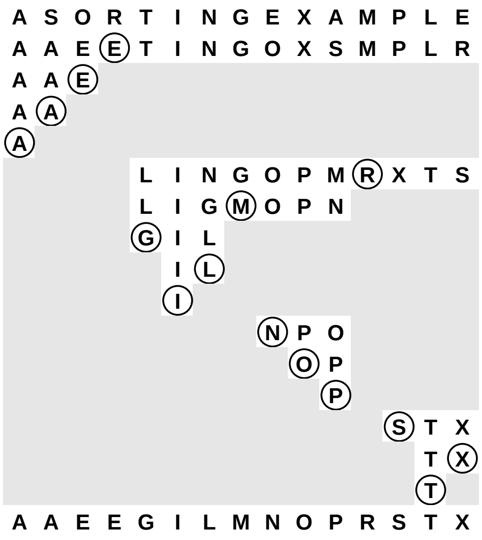
COS 226 Lecture 3: Quicksort

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

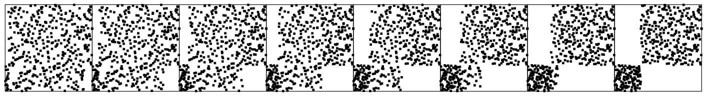
Quicksort example



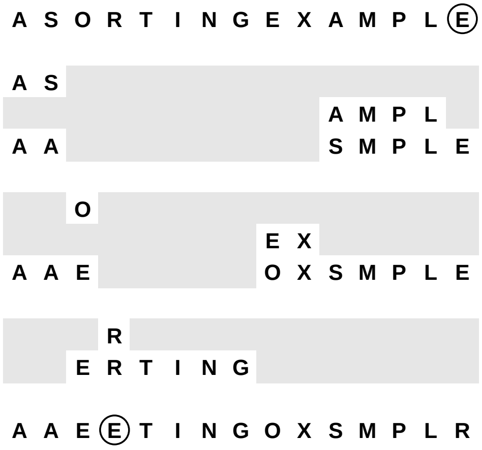
```
A S O R T I N G E X A M P L E
A A E Ⓔ T I N G O X S M P L R
A A Ⓔ
A Ⓐ
Ⓐ
        L I N G O P M Ⓡ X T S
        L I G Ⓜ O P N
        Ⓖ I L
          I Ⓛ
          Ⓘ
                Ⓝ P O
                  Ⓞ P
                    Ⓟ
                        Ⓢ T X
                          T Ⓧ
      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

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



Partitioning example



```
A S O R T I N G E X A M P L Ⓔ

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 Ⓔ T I N G O X S M P L R
```

Partitioning implementation

```
int partition(Item a[], int l, int r)
{ int i, j; Item 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(Item a[], int l, int r)
{ int i;
   if (r > l)
      {
         i = partition(a, 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(Item 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); }
      }
```

Analysis of Quicksort

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

Worst case analysis

Number of comparisons in the worst case
- $N + (N-1) + (N-2) + \ldots = N(N-1)/2$

Worst case files
- already sorted (!)
- reverse order
- all equal? (stay tuned)

Total time proportional to N^2

No better than elementary sorts?

Fix: use a random partitioning element
- ``guarantees`` fast performance

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

$$
\begin{aligned}
C(N) = N+1 \;&+\; (C(1) + C(N-1))/N \\
&+\; (C(2) + C(N-2))/N \\
&\ldots \\
&+\; (C(N-1) + C(1))/N
\end{aligned}
$$

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

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

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

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

$$
\begin{aligned}
C(N)/(N+1) &= C(N-1)/N + 2/(N+1) \\
&= 2(\,1 + 1/2 + 1/3 + \ldots\ 1/(N+1)\,) \\
&= 2 \ln N + (\text{small error term})
\end{aligned}
$$

THM: Quicksort uses about 2N ln N comparisons

Empirical analysis

Use profiler

Inner loop
- look for highest counts
- is every line of code there necessary?

Verify analysis
- are counts in predicted range?

Streamline program by iterating process

Quicksort profile

```
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>}
```

Ex: another partitioning 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-sample

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

Insertion sort small subfiles

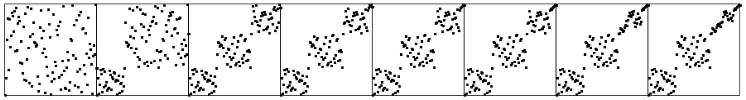
- even Quicksort has too much overhead
  for files of a few elements
- use insertion sort for tiny files
  (can wait until the end)

Optimize parameters

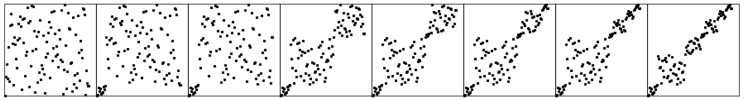
- median of 3 elements
- cut to insertion sort for < 10 elements

Improvements to Quicksort (examples)

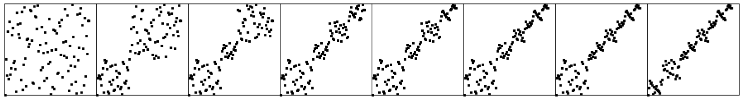
Standard



Cutoff for small subfiles



Median-of-three



Selection

Use partitioning to find the k-th smallest element

- (don't need to sort the whole file)

```
select(Item a[], int l, int r, int k)
  { int i;
    if (r <= l) return;
    i = partition(a, l, r);
    if (i > k) select(a, l, i-1, k);
    if (i < k) select(a, i+1, r, k);
  }
```

Ex: to find median

```
    select(a, l, r, (l+r)/2);
```

Also puts k smallest elements in first k positions

Running time is LINEAR on the average

linear time guarantee possible?

- old theorem says yes; not useful in practice
- randomized guarantee just about as good

Equal keys

Equal keys can adversely affect performance

One key value (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 (!!)

Two distinct key values
- reduces to above case for one subfile
- better to complete sort with one partition

    stop right ptr on 0; stop left ptr on 1; exchange

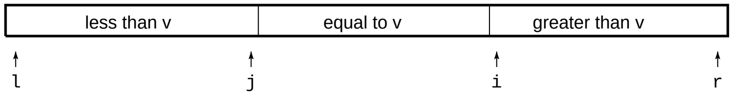
Several distinct key values
- reduces to above cases

Serious performance bug in widely-used implementations

Three-way partitioning problem

Natural way to deal with equal keys

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

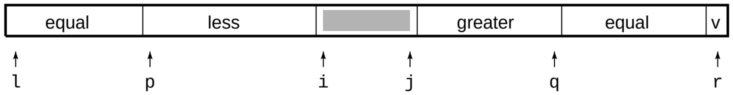


Dutch National Flag problem
- Not easy to implement efficiently (try it!)
- Not done in practical sorts before mid-1990s

Three-way partitioning solution

Four-part partition
- some elements between i and j equal to v
- no larger element left of i
- no smaller element right of j
- more elements between i and j equal to v

Swap equal keys into center



All the right properties
- easy to implement
- linear if keys all equal
- no extra cost if no equal keys

Three-way partitioning implementation

```
 void quicksort(Item a[], int l, int r)
 {
 int i, j, k, p, q; Item v;
 if (r <= l) return;
 v = a[r]; i = l-1; j = r; p = l-1; q = 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]);
    if (eq(a[i],v)) { p++; exch(a[p],a[i]); }
    if (eq(v,a[j])) { q--; exch(a[q],a[j]); }
 }
 exch(a[i], a[r]); j = i-1; i = i+1;
 for (k = l  ; k < p; k++, j--) exch(a[k], a[j]);
 for (k = r-1; k > q; k--, i++) exch(a[k], a[i]);
 quicksort(a, l, j);
 quicksort(a, i, r);
}
```

Significance of three-way partitioning

Equal keys omnipresent in applications
- ex: sort population by age
- ex: sort job applicants by college attended

Purpose of sort: bring records with equal keys together

Typical application
- Huge file
- Small number of key values

randomized 3-way Quicksort is LINEAR time (try it!)

THM: Quicksort with 3-way partitioning is OPTIMAL
Proof: (beyond the scope of 226) ties cost to entropy

[this fundamental fact was not known until 2000!]