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
Quick sort example

Sorting example:

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

L I N G O P M R X T S
L I G M O P N
G I L
L I
I

N O P O
O P
P

S T X
T X
T
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

ASORTINGEXAMPLE

ASAMPLE

AASAMPLE

OXEXSAMPLE

AAXSAMPLE

RETING

AAXETINGOXSAMPLE
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

```c
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

```c
#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
- \( N + (N-1) + (N-2) + ... = N(N-1)/2 \)

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
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 comparisons satisfies

\[ C(N) = N+1 + \frac{(C(1) + C(N-1))}{N} \]
\[ + \frac{(C(2) + C(N-2))}{N} \]
\[ ... \]
\[ + \frac{(C(N-1) + C(1))}{N} \]

\[ C(N) = N+1 + 2\left( \frac{C(1) + C(2) + ... + C(N-1)}{N} \right) \]

\[ NC(N) = N(N+1) + 2\left( C(1) + C(2) + ... + C(N-1) \right) \]

\[ NC(N) - (N-1)C(N-1) = 2N + 2C(N-1) \]

\[ NC(N) = (N+1)C(N-1) + 2N \]

\[ \frac{C(N)}{(N+1)} = \frac{C(N-1)}{N} + \frac{2}{(N+1)} \]
\[ = 2\left( 1 + \frac{1}{2} + \frac{1}{3} + ... \frac{1}{(N+1)} \right) \]
\[ = 2 \ln N + \text{(small error term)} \]

**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(int a[], int l, int r)
{
    int v, i, j, t;
    if (r > l)
    {
        v = a[r];
        i = l-1; j = r;
        for ;
            while (a[++i] < v);
            while (a[--j] > v);
            if (i >= j) break;
        t = a[i]; a[i] = a[j]; a[j] = t;
    }
    t = a[i]; a[i] = a[r]; a[r] = t;
    quicksort(a, l, i-1);
    quicksort(a, i+1, r);
}
Ex: another partitioning method

(detailed justification omitted)

```c
quicksort(int a[], int l, int r)
{
  int v, i, k, t;
  if (r <= l) return;
  v = a[l]; k = l;
  for (i=l+1; i<=r; i++)
    if (a[i] < v)
      { t = a[i]; a[i] = a[++k]; a[k] = t; }
  t = a[k]; a[k] = a[l]; a[l] = t;
  quicksort(a, l, k-1);
  quicksort(a, k+1, r);
}
```

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
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 paritioning 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

<table>
<thead>
<tr>
<th>less than v</th>
<th>equal to v</th>
<th>greater than v</th>
</tr>
</thead>
<tbody>
<tr>
<td>↑</td>
<td>↑</td>
<td>↑</td>
</tr>
<tr>
<td>l</td>
<td>j</td>
<td>i</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r</td>
</tr>
</tbody>
</table>

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

\[
\begin{array}{cccccc}
\text{equal} & \text{less} & \text{greater} & \text{equal} & \text{v} \\
\uparrow & \uparrow & \uparrow & \uparrow & \uparrow \\
\downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\
1 & p & i & j & q & r
\end{array}
\]

All the right properties

- easy to implement
- linear if keys all equal
- no extra cost if no equal keys
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);
}
PROBLEM:
- long key strings costly to compare when they differ only at the end
- [this is the common case!]

SOLUTION:
- Use three-way partitioning on key characters
- Recurse and pass current char index
- (to be continued: this is a radix sort)
## String sort example

<table>
<thead>
<tr>
<th>actinian</th>
<th>coenobite</th>
<th>actinian</th>
</tr>
</thead>
<tbody>
<tr>
<td>jeffrey</td>
<td>conelrad</td>
<td>bracteal</td>
</tr>
<tr>
<td>coenobite</td>
<td>actinian</td>
<td>coenobite</td>
</tr>
<tr>
<td>conelrad</td>
<td>bracteal</td>
<td>conelrad</td>
</tr>
<tr>
<td>secureness</td>
<td>secureness</td>
<td>cumin</td>
</tr>
<tr>
<td>cumin</td>
<td>dilatedly</td>
<td>chariness</td>
</tr>
<tr>
<td>chariness</td>
<td>inkblot</td>
<td>centesimal</td>
</tr>
<tr>
<td>bracteal</td>
<td>jeffrey</td>
<td>cankerous</td>
</tr>
<tr>
<td>displease</td>
<td>displease</td>
<td>circumflex</td>
</tr>
<tr>
<td>millwright</td>
<td>millwright</td>
<td>millwright</td>
</tr>
<tr>
<td>repertoire</td>
<td>repertoire</td>
<td>repertoire</td>
</tr>
<tr>
<td>dourness</td>
<td>dourness</td>
<td>dourness</td>
</tr>
<tr>
<td>centesimal</td>
<td>southeast</td>
<td>southeast</td>
</tr>
<tr>
<td>fondler</td>
<td>fondler</td>
<td>fondler</td>
</tr>
<tr>
<td>interval</td>
<td>interval</td>
<td>interval</td>
</tr>
<tr>
<td>reversionary</td>
<td>reversionary</td>
<td>reversionary</td>
</tr>
<tr>
<td>dilatedly</td>
<td>cumin</td>
<td>secureness</td>
</tr>
<tr>
<td>inkblot</td>
<td>chariness</td>
<td>dilatedly</td>
</tr>
<tr>
<td>southeast</td>
<td>centesimal</td>
<td>inkblot</td>
</tr>
<tr>
<td>cankerous</td>
<td>cankerous</td>
<td>jeffrey</td>
</tr>
<tr>
<td>circumflex</td>
<td>circumflex</td>
<td>displease</td>
</tr>
</tbody>
</table>