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

Quicksort example

Partitioning example

ASORTINGEXAMPLE
AAEETINGOXSPLR

ASAMPLE
AA

OEX
AAXE

RETING
AAEETINGOXSPLR
Partitioning implementation

```c
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

```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
- \( \text{cost} \times \text{frequency} \)
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
- \( N + (N-1) + (N-2) + \ldots = \frac{N(N-1)}{2} \)

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 comparisons satisfies
\[
C(N) = \frac{N}{2} + \frac{C(1) + C(N-1)}{N} + \frac{C(2) + C(N-2)}{N} + \ldots + \frac{C(N-1) + C(1)}{N}
\]

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

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

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

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

\[
= 2 \left( \frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{N+1} \right)
\]

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

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 profile

```c
quicksort(int a[], int l, int r)
{
    int v, i, j, t;
    if (r > l)
    {
        v = a[r];
        i = l-1; j = r;
        while (a[++i] < v);
        while (a[--j] > v);
        if (i >= j) break;
        t = a[i]; a[i] = a[j]; a[j] = 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 (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)

```c
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

```c
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

---

Median-of-sample
- partitioning element closer to center
- estimate median with median of sample
- number of comparisons close to \( N \log N \)
- 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
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

| equal | less | greater | equal | $v$
|-------|------|---------|-------|---
|   1   |   p  | 1       |  j    | q  |
|  l    |  p   | i       |  j    | q  |
|       |      |         |       | r  |

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

Three-way partitioning implementation

```c
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!]