COS 226 Lecture 6: Radix sorting

• Bits and digits
• Binary Quicksort
• MSD radix sort
• Three-way radix Quicksort
• LSD radix sort
• Sorting in linear time

6.1 Bits and digits

Extracting bits is easy in C

Radix: base of number system

Power of 2 radix: groups of bits
• binary (radix-2): 1 bit at a time
• hexadecimal (radix-16): 4 bits at a time
• ascii (radix-256): 8 bits at a time

| bin   | 01100001011100010011001101100100 |
| hex   | 6 1 6 2 6 3 6 4 |
| ascii | a b c d |

6.2 Extracting digits with macros

#define bitsword 32
#define bitsbyte 8
#define bytesword 4
#define R (1 << bitsbyte)
#define digit(A, B) ((A >> (bitsword-(B+1)*bitsbyte)) & (R-1))

Ex: Single-byte access: bitsbyte = 8
x = 0x61626364
digit(x, 2) = (x >> 8) & 255 = c
0110 0001 0110 0010 0110 0011 0110 0100 x
0000 0000 0110 0001 0110 0010 0110 0011 x >> 8
0000 0000 0000 0000 0000 0000 1111 1111 255 (R-1)
0000 0000 0000 0000 0000 0000 0110 0011 c

Ex: Single-bit access: bitsbyte = 1
digit(x, 11) = (x >> 20) & 1 = 0
0110 0001 0110 0010 0110 0011 0110 0100 x
0000 0000 0000 0000 0000 0110 0001 0110 0110 x >> 20
0000 0000 0000 0000 0000 0000 0000 0001 1 (R-1)

6.3 Binary Quicksort

Partition file into two pieces
• all keys with first bit 0
• all keys with first bit 1
Sort two pieces recursively

Equivalent to partitioning on the VALUE 2^(bitsword-w+1)
• instead of some key in the file.

Bad partition if all keys have same leading bit
• one subfile of size N
• one empty subfile
• BUT keys one bit shorter

Worst case: one pass per key bit

#define bitsword 32
#define bitsbyte 8
#define bytesword 4
#define R (1 << bitsbyte)
#define digit(A, B) ((A >> (bitsword-(B+1)*bitsbyte)) & (R-1))

Ex: Single-byte access: bitsbyte = 8
x = 0x61626364
digit(x, 2) = (x >> 8) & 255 = c
0110 0001 0110 0010 0110 0011 0110 0100 x
0000 0000 0110 0001 0110 0010 0110 0011 x >> 8
0000 0000 0000 0000 0000 0000 1111 1111 255 (R-1)
0000 0000 0000 0000 0000 0000 0110 0011 c

Ex: Single-bit access: bitsbyte = 1
digit(x, 11) = (x >> 20) & 1 = 0
0110 0001 0110 0010 0110 0011 0110 0100 x
0000 0000 0000 0000 0000 0110 0001 0110 0110 x >> 20
0000 0000 0000 0000 0000 0000 0000 0001 1 (R-1)
Binary Quicksort code

```c
quicksortB(int a[], int l, int r, int w)
{
    int i = l, j = r;
    if (r <= l || w > bitsword) return;
    while (j != i)
    {
        while (digit(a[i], w) == 0 && (i < j)) i++;
        while (digit(a[j], w) == 1 && (j > i)) j--;
        exch(a[i], a[j]);
    }
    if (digit(a[r], w) == 0) j++;
    quicksortB(a, l, j-1, w+1);
    quicksortB(a, j, r, w+1);
}
```

Binary Quicksort issues

Problems:
- leading 0 bits
- cost of inner loop
  (could be advantage if carefully done)

Worst case: all keys equal
- 32N passes on a 32-bit machine
- 64N passes on a 64-bit machine
Good way to avoid quadratic worst case of quicksort

Random bits?
- should sort out after lgN bits examined
Nonrandom bits?
- take bigger chunks

MSD radix sort

Partition file into R buckets
- all keys with first byte 0
- all keys with first byte 1
- all keys with first byte 2
  ...
- all keys with first byte R-1
Sort R pieces recursively

Take R=2^bitsbyte

Tradeoff
- large R: space for buckets (too many empty buckets)
- small R: too many passes (too many keys per bucket)

Upper bound on running time: (bytesword)*(N + R)
(Worst case: all keys equal)
MSD radix sort example

now ace ace ace
for ago ago ago
tip and and and
ilk bet bet bet
dim cab cab cab
tag caw caw caw
jot cue cue cue
sob dim dim dim
nob dug dug dug
sky egg egg egg
hut few few few
ace fee fee few
bet bet bet bet
men gig gig gig
egg hut hut hut
few ilk ilk ilk
jay jam jay jam
owl jay jay jay
joy jot jot jot
rap joy joy joy
gig men men men
wee now now now
was nob nob nob
job cab owl owl
wad rap rap rap
caw sob sky sky
cue sky sob sob
fee tip tag tag
tap tag tap tap
ago tap tar tar
tar tar tip tip
jam wee wad wad
dug was was was
and wad wee wee

Key-indexed counting example

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>6</td>
<td>4</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>6</td>
<td>10</td>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Key-indexed counting

Basis for radix sorts: sort file of keys with R values
- count number of keys with each value
- take sums to turn counts into indices
- move keys to auxiliary array using indices

Need one counter for each different key value

```c
void keycount(int a[], int l, int r)
{ int i, j, cnt[R+1];
  int b[maxN];
  for (j = 0; j < R; j++) cnt[j] = 0;
  for (i = l; i <= r; i++) cnt[a[i]+1]++;
  for (j = 1; j < R; j++) cnt[j] += cnt[j-1];
  for (i = l; i <= r; i++)
    b[cnt[a[i]]++] = a[i];
  for (i = l; i <= r; i++)
    a[i] = b[i];
}
```

MSD radix sort

Three changes to key-indexed counting code

1. Modify key access to extract bytes
   - start with Most Significant Digit
   - divides files into R subfiles

2. Sort the R subfiles recursively
   - but use insertion sort for small files

3. To handle variable-length keys terminated with 0
   - remove test for end of key
   - remove recursive call corresponding to 0

Most important keys to good performance:
- fast byte extraction
- cutoff to insertion sort
#define bin(A) l+count[A]

void radixMSD(Item a[], int l, int r, int w)
{
    int i, j, count[R+1];
    if (w > bytesword) return;
    if (r-l <= M) { insertion(a,l,r); return; }
    for (j = 0; j < R; j++) count[j] = 0;
    for (i = l; i <= r; i++)
        count[digit(a[i], w) + 1]++;
    for (j = 1; j < R; j++)
        count[j] += count[j-1];
    for (i = l; i <= r; i++)
        b[l+count[digit(a[i], w)]] = a[i];
    for (i = l; i <= r; i++) a[i] = b[i];
    radixMSD(a, l, bin(0)-1, w+1);
    for (j = 0; j < R-1; j++)
        radixMSD(a, bin(j), bin(j+1)-1, w+1);
}

void radixLSD(Item a[], int l, int r)
{
    int i, j, w, count[R+1];
    for (w = bytesword-1; w >= 0; w--)
    {
        for (j = 0; j < R; j++) count[j] = 0;
        for (i = l; i <= r; i++)
            count[digit(a[i], w)]++;
        for (i = l; i <= r; i++)
            b[count[digit(a[i], w)]] = a[i];
        radixLSD(a, l, bin(0)-1, w+1);
        for (j = 0; j < w; j++)
            radixLSD(a, bin(j), bin(j+1)-1, w+1);
    }
}

### MSD radix sort code

Each pass ALWAYS takes time proportional to N+R
- Initialize the buckets
- Scan the keys

Ex: (ASCII bytes) R = 256
- 100 times slower than insertion sort for N = 2

Ex: (UNICODE) R = 65536
- 30,000 times slower than insertion sort for N = 2

Too slow for small files
Recursive program will call itself for a huge number of small files

Solution: cutoff to insertion sort
LSD radix sort example

```
now sob cab ace
for nob wad ago
tip cab tag and
ilk wad jam bet
dim and rap cab
tag ace tap caw
jot wee tar cue
sob cue wad dim
nob fee cab dug
sky tag raw egg
hut egg jay fee
ace gig fee few
bet dug wee for
men ilk fee gig
egg owl men hut
few dim bet ilk
jay jam few jay
joy ago ago jot
rap tip gig joy
gig rap dim men
wee tap tip nob
was fgr sky now
cab tar ilk owl
wad was and rap
tap jot sob raw
caw hut rob sky
cue bet for sob
fee you jot tag
raw now you tap
ago few fow tar
tar caw jay tip
jam raw cay wad
dug sky dug was
you jay hut wee
and jay owl you
```

Two proofs for LSD radix sort

**Left-right**
- If two keys differ on first bit
  - 0-1 sort puts them in proper relative order
- If two keys agree on first bit
  - Stability keeps them in proper relative order

**Right-left**
- If the bits not yet examined differ
  - Doesn’t matter what we do now
- If the bits not yet examined agree
  - Later pass won’t affect their order

Binary LSD radix sort example

```
A 000001 R 10000
S 010101 T 00101
O 011101 N 01111
P 01000 L 100101
I 010101 L 101001
N 01110 A 00001
G 00111 S 10001
E 01101 O 01111
M 00000 T 00100
X 10000 I 01001
A 00001 Q 00111
M 01101 E 01011
P 00000 A 00000
L 00100 M 01100
E 00101 O 01111
G 00001 X 00000
```

Cannot use Quicksort-style partitioning
- 0-1 sort has to be stable
- Stable inplace 0-1 sort? (possible, but not easy)

Linear sorting method

LSD radix sort!
- To sort N 64-bit keys take byte=16
  - 4N steps, linear extra memory (plus 2^16)

Does not violate N lg N lower bound because
- Comparisons are not used

LSD radix sort liabilities
- Inner loop has a lot of instructions
- Accesses memory “randomly”
- Wastes time on low-order bits

Therefore, use just “enough” bits
LSD-MSD hybrid

MSD radix sort also linear
Use LSD-MSD hybrid for random keys
  (assume fixed-size keys)
  use \((\log N)/2 < \text{bits byte} < \log N\)

Three passes
  - LSD radix sort on 2nd byte
  - LSD radix sort on 1st byte
  - insertion sort to clean up

Recursive structure of MSD radix sort

Tree structure to describe recursive call
Paths in tree give keys

Problem: algorithm touches empty nodes

Tree can be as much as \(M\) times bigger than they seem

---

Sorting strings

PROBLEM:
  - long key strings costly to compare
    when they differ only at the end
  - [this is the common case!]

SOLUTION: 3-way radix Quicksort
  - Use three-way partitioning on key characters
  - Recurse and pass current char index

---

3-way radix Quicksort partitioning

| 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 |
3-way radix Quicksort code

#define ch(A) digit(A, D)
void quicksortX(Item a[], int l, int r, int D)
{
  int i, j, k, p, q; int v;
  if (r-l <= M) { insertion(a, l, r); return; }
  v = ch(a[r]); i = l-1; j = r; p = l-1; q = r;
  while (i < j)
  {
    while (ch(a[++i]) < v);        
    while (v < ch(a[--j])) if (j == l) break;
    if (i > j) break;
    exch(a[i], a[j]);
    if (ch(a[i])==v) { p++; exch(a[p], a[i]); }
    if (v==ch(a[j])) { q--; exch(a[j], a[q]); }
  }
  if (p == q)
    { if (v != '\0') quicksortX(a, l, r, D+1); return; }
  if (ch(a[i]) < v) i++;
  for (k = l; k <= p; k++, j--) exch(a[k], a[j]);
  for (k = r; k >= q; k--, i++) exch(a[k], a[i]);
  quicksortX(a, l, j, D);
  if ((i == r) && (ch(a[i]) == v)) i++;
  if (v != '\0') quicksortX(a, j+1, i-1, D+1);
  quicksortX(a, i, r, D);
}

3-way radix Quicksort example

now gig ace ago a|go
for for bet bet a|ce
tip dug dug and a|nd
ilk ilk cab ace b|et
dim dim dim c|ab
tag ago ago caw
jot and and c|ue
sob fee egg egg
nob cue cue dug
sky caw caw dim
hut hut fee
ace ace for
bet bet few
men cab ilk
egg egg gig
few few hut
jay jay jam
owl jot jay
joy joy joy
rap jam jot
gig owl owl men
wee wee now owl
was was nob nob
men men men
wad wad rap
ca|w sky sky sky sky
cue nob was tip sob
fee sob sob sob t|ip tar
tap tap tap tap tap
tag tag tag tag tag
tag tar tar tar tar tip
tap tip ti|p was
and now wee wee
jam rap wad wad

Another sublinear sort

Three-way radix quicksort is SUBLINEAR

N records with w-byte keys
- Bytes of data: Nw
- Bytes examined by sort: 2 N ln N
Ex: 100000 keys, 100 bytes per key
- 10 million bytes of data
- algorithm examines 2.3 million bytes
  1/5 of the data
Corresponds to collapsing null links in MSD trees