Outline

Today:

- Quick sort and improvements
- Distribution sort
Quick Sort

Like mergesort, it is a divide-and-conquer algorithm

Mergesort: easy division, complex combination

mergesort(left half)
mergesort(right half)
merge(left, right)

Quicksort: complex division, easy combination

partition(left, right)
quicksort(left partition)
quicksort(right partition)
Quick Sort

Algorithm (see Fig. 15.4):

- choose an element of list as pivot
- put all elements < pivot to the left of pivot
- put all elements ≥ pivot to the right of pivot
- move pivot to its correct place on the list
- sort left and right subarrays recursively (not including pivot)

```c
void qsort(int *a, left, right)
{
    int split; // index of where the pivot is

    if (left >= right) return;
    split = partition(a, left, right);
    qsort(a, left, split-1);
    qsort(a, split+1, right);
}
```

Sugih Jamin (jamin@eecs.umich.edu)
Partition [Lomuto]

int partition(int *a, left, right)
{
    int i, j, pivot;

    pivot = a[left];
    for (i = left, j = i+1; j < right; j++) {
        /* continue to look for elt < pivot */
        if (a[j] < pivot) {
            i++;
            /* swap with first >= elt */
            swap(a[i], a[j]);
        }
    }

    /* put pivot in place,
       swap with last elt < pivot */
    swap(a[left], a[i]);
    return i;
}

Running time:
Running Time of Quicksort

Think of it as building a binary tree

Best case:
- partition: $O(N)$
- height of tree: $O(\log N)$
- total: $O(N \log N)$

Worst case: when array is pre-sorted
- partition: $O(N)$
- height of tree: $O(N)$
- total: $O(N^2)$

Average case: $O(N \log N)$
Pivot Selection

Median-of-three:

- ideally: pivot is the median of the elements’ values, but to find median, need to sort!
- heuristic: choose the middle value of the first, last, and middle elements as pivot

Randomized: choose a random element as pivot, running time $O(N \log N)$, with high probability
Quicksort is In-place, but . . .

Recursion uses up stack space
Worst case needs $\Theta(N)$ stack space

Small list:

- do insertion sort if array size is below threshold
- but quicksort works by reducing array size!
- terminate quick sort when array is below threshold, leave unsorted subarrays in place
- subarrays are sorted relative to each other
- do insertion sort on the whole array as last step

Quicksort is not stable
Tail-Recursion Removal

Tail recursion:
• when the last statement of a recursive function calls itself
• can be replaced by an iterative loop

Examples:

```c
int count(link *list)
{
    if (!list) return 0;
    return 1+count(list->next);
}

void walk(link *list)
{
    if (!list) return;
    visit(list);
    walk(list->next);
}

void reverse_walk(link *list)
{
    if (!list) return;
    reverse_walk(list->next);
    visit(list);
}
```

Sugih Jamin (jamin@eecs.umich.edu)
Tail-recursion Removed Quicksort

Observations:

- quicksort calls itself twice
- we can remove the last call
- the first call still uses up stack space
- the order of doing left partition first or right partition first does not matter

Call quicksort recursively only on the smaller partition
Sort the larger partition iteratively
Improved Quicksort

```c
void
qsort(int *a, left, right)
{
    qsort_tr(a, left, right);
    isort(a, right-left);
}

void
qsort_tr(int *a, left, right)
{
    while (right-left > smallsize) {
        split = partition(a, left, right);
        if (split < ((left+right) >> 1)) {
            qsort_tr(a, left, split-1);
            left = split+1; /* right stays */
        } else {
            qsort_tr(a, split+1, right);
            right = split-1; /* left stays */
        }
    }
    return;
}
```

Sugih Jamin (jamin@eecs.umich.edu)
## Running Times of Sorting Algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Best</th>
<th>Average</th>
<th>Worst</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear insertion</td>
<td>$O(n)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Binary insertion</td>
<td>$O(n \log n)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Bubble</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Quicksort</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Straight selection</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Heapsort</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>Mergesort</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
</tr>
</tbody>
</table>

Sugih Jamin (jamin@eecs.umich.edu)
Intuition on Sorting Algorithms Performance

How can mergesort and quicksort achieve $O(n \log n)$ when insertion sort and selection sort run at $O(n^2)$?

Mergesort and quicksort move elements far distances, correcting multiple inversions at a time.

Why is quicksort worst-case $O(n^2)$ while mergesort has no such problem?

The choice of pivot determines size of partitions, whereas mergesort cuts array in half everytime.

Is $O(n \log n)$ the best we can do?

Sugih Jamin (jamin@eecs.umich.edu)
Decision Tree and The Information Theoretic Lower Bound

Is $O(n \log n)$ the best we can do?
Yes, if we use binary comparison on the key values

$N$ elements in a decision tree result in $N!$ permutations of the elements at the leaves (see Preiss Fig. 15.11). Let $h$ be the height of tree:

\[
2^h = N!
\]

\[
h = \lceil \log N! \rceil
\]

\[
\geq \log_2 N!
\]

\[
\geq \log_2 (1 \cdot 2 \cdot 3 \ldots \cdot N)
\]

\[
\geq \sum_{i=1}^{N} \log_2 i
\]

\[
\geq \log_2 1 + \log_2 2 + \log_2 3 + \ldots + \log_2 \frac{N}{2} + \ldots + N
\]

\[
\geq \frac{N}{2} \log_2 \frac{N}{2}
\]

\[= \Omega(N \log N)\]
Distribution/Digital Sort

To escape the Information Theoretic Lower bound, instead of comparing keys, use them as data:

- bucket/counting sort
- radix sort
- radix exchange sort
Bucket/Counting Sort

- if keys are small non-negative integers, use them as indices into a count/bucket array
- example: to sort $a[0,\ldots,n]$, first allocate $bucket[0,\ldots,k]$, where $k$ is the maximum key value
- to sort, simply increment: $bucket[a[0]], bucket[a[1]],\ldots, bucket[a[n]]$
- walk down $bucket[i]$ $0 \leq i \leq k$ and for each $i$, print $i$ out $bucket[i]$ times

Running time: $O(n)$, but $k$ cannot be too large

Instead of simple integer, each element can be a (pointer to) record, then instead of incrementing the count of each bucket, **distribute** the records into their appropriate buckets

Bucket/counting sort is stable if the distribution step is stable
Radix Sort

Useful when key values are large but consist of base/radix (e.g., digits or chars) that each can be sorted using bucket sort
Examples:

- “hello” is a string of 5 characters, radix-26
- “429” is a string of 3 digits, radix-10

Algorithm:

- do bucket sort on the least significant (rightmost) radix
- then sort the resulting list on the next significant radix
- algorithm works only if the bucket sort used is stable
- not in-place

Example: sort CX AX BZ BY AZ BX AY

- first pass: CX AX BX BY AY BZ AZ
- second pass: AX AY AZ BX BY BZ CX
Radix Sort

Running time:

- let $d$ be the string length (assume equal length strings), $N$: number of strings, and $R$ the radix size
- algorithm makes $d$ passes over the strings: $O(dN)$
- at each pass, the $i$-th element of the strings are sorted: $O(dR)$
- total running time: $O(d(R + N)) = O(N)$ for $d$ and $R$ small
- but note that if keys are treated as binary numbers and radix-2 is used, algorithm degenerates into $O(N \log N)$ (each binary number becomes a binary comparison, for 32-bit key, $N = 2^{32}$ and $d = 32$ is $d = \log N$)

Sugih Jamin (jamin@eecs.umich.edu)
Radix Exchange Sort

Radix sort sorts from the least significant radix first

Can you sort from most significant radix (MSR) first?
Yes, by doing a form of quicksort (running time $O(N \log N)$)

For example:

• assume radix-2 and strings of length $N$
• move all strings starting with MSR = ’0’ to the left
• move all strings starting with MSR = ’1’ to the right
• first pivot is 1000...0, i.e., 1 followed by $(N - 1)$ 0’s
• next recursively sort both partitions with the next significant bit = 1
• etc.