Elementary Sorting Methods

Bùi Tiến Lên

2024

KHOA CÔNG NGHỆ THÔNG TIN
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

Contents

Introduction

Concept 1

Sorting is a fundamental algorithm design problem. Many efficient algorithms use sorting as a subroutine, because it is often easier to process data if the elements are in a sorted order.

- The basic problem in sorting is as follows: Given an array that contains n elements (keys), our task is to sort the elements in increasing order. For example, the array

1	3	8	2	9	2	5	6

will be as follows after sorting:

1	2	2	3	5	6	8	9

Introduction (cont.)

Concept 2

An **inversion** is a pair of keys that are out of order in the array

- For instance, an array {E, X, A, M, P, L, E} has 11 inversions: E-A, X-A, X-M, X-P, X-L, X-E, M-L, M-E, P-L, P-E, and L-E.

Concept 3

If the number of inversions in an array is less than a constant multiple of the array size, we say that the array is **partially sorted**.

Introduction (cont.)

Concept 4

A sorting method is **stable** if it preserves the relative order of equal keys in the array

sorted by time		sorted by location (not stable)		sorted by location (stable)	
Chicago	09:00:00	Chicago	09:25:52	Chicago	09:00:00
Phoenix	09:00:03	Chicago	09:03:13	Chicago	09:00:59
Houston	09:00:13	Chicago	09:21:05	Chicago	09:03:13
Chicago	09:00:59	Chicago	09:19:46	Chicago	09:19:32
Houston	09:01:10	Chicago	09:19:32	Chicago	09:19:46
Chicago	09:03:13	Chicago	09:00:00	Chicago	09:21:05
Seattle	09:10:11	Chicago	09:35:21	Chicago	09:25:52
Seattle	09:10:25	Chicago	09:00:59	Chicago	09:35:21
Phoenix	09:14:25	Houston	09:01:10	Houston	09:00:13
Chicago	09:19:32	Houston	09:00:13	Houston	09:01:10
Chicago	09:19:46	Phoenix	09:37:44	Phoenix	09:00:03
Chicago	09:21:05	Phoenix	09:00:03	Phoenix	09:14:25
Seattle	09:22:43	Phoenix	09:14:25	Phoenix	09:37:44
Seattle	09:22:54	Seattle	09:10:25	Seattle	09:10:11
Chicago	09:25:52	Seattle	09:36:14	Seattle	09:10:25
Chicago	09:35:21	Seattle	09:22:43	Seattle	09:22:43
Seattle	09:36:14	Seattle	09:10:11	Seattle	09:22:54
Phoenix	09:37:44	Seattle	09:22:54	Seattle	09:36:14

no longer sorted by time

still sorted by time

Introduction (cont.)

Concept 5

An **adaptive sorting algorithm** is a type of sorting algorithm that takes advantage of existing order or structure in the input data to improve its performance.

Concept 6

A **non-adaptive sorting algorithm** is a type of sorting algorithm that does not take advantage of any existing order or structure in the input data.

Sorting Algorithms

$O(n^2)$ algorithms

- Selection Sort
- Insertion Sort
- Bubble Sort

$O(n^k)$ algorithms

- Shell Sort

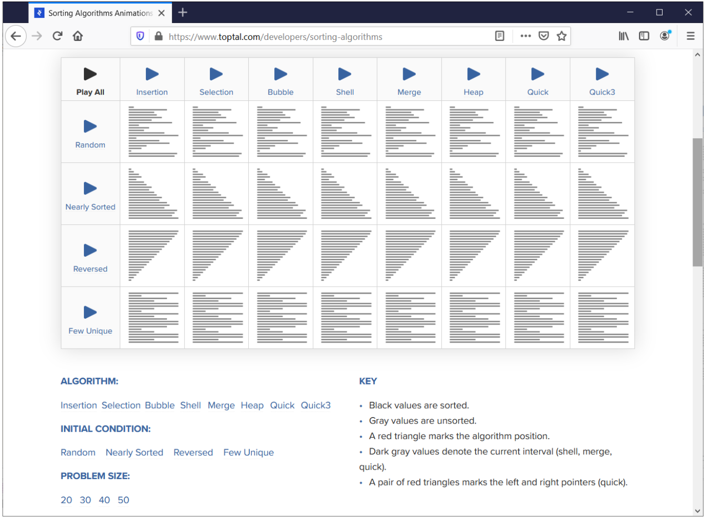
$O(n \log n)$ algorithms

- Heap Sort
- Merge Sort
- Quick Sort

$O(n)$ algorithms

- Radix Sort
- Counting Sort

Visualizing Sorting Algorithms



Selection Sort

Selection Sort

Idea

Assume that the array $\boldsymbol{a}$ is composed of two parts: the left part $\boldsymbol{s}$ is sorted and the right part $\boldsymbol{u}$ is unsorted

1. $\boldsymbol{s} = \emptyset$ and $\boldsymbol{u} = \boldsymbol{a}$
2. Find the smallest element x of the part $\boldsymbol{u}$
3. Remove x from $\boldsymbol{u}$
4. Append x to $\boldsymbol{s}$

Repeat the actions 2-4 until $\boldsymbol{u}$ is empty

Implementation

```
template <class Item>
void selection(Item a[], int l, int r) {
  for (int i = l; i < r; i++) {
    int min = i;
    for (int j = i + 1; j <= r; j++)
      if (a[j] < a[min]) min = j;
    swap(a[i], a[min]);
  }
}
```

Illustration

- Trace (right before the call to swap())

a[]

i	min	0	1	2	3	4	5	6	7	8	9	10
		S	O	R	T	E	X	A	M	P	L	E
0	6	S	O	R	T	E	X	A	M	P	L	E
1	4	A	O	R	T	E	X	S	M	P	L	E
2	10	A	E	R	T	O	X	S	M	P	L	E
3	9	A	E	E	T	O	X	S	M	P	L	R
4	7	A	E	E	L	O	X	S	M	P	T	R
5	7	A	E	E	L	M	X	S	O	P	T	R
6	8	A	E	E	L	M	O	S	X	P	T	R
7	10	A	E	E	L	M	O	P	X	S	T	R
8	8	A	E	E	L	M	O	P	R	S	T	X
9	9	A	E	E	L	M	O	P	R	S	T	X
10	10	A	E	E	L	M	O	P	R	S	T	X
		A	E	E	L	M	O	P	R	S	T	X

Analysis

Theorem 1

*Selection sort uses $\sim N^2/2$ **compares** and N **exchanges** to sort an array of length N.*

Analysis of selection sort for the input size of N (the number of keys)

- Time complexity:

best case	?
average case	?
worst case	?

- Space complexity: $O(1)$
- Stability: No

Insertion Sort

Insertion Sort

Idea

Assume that the array $\boldsymbol{a}$ is composed of two parts: the left part $\boldsymbol{s}$ is sorted and the right part $\boldsymbol{u}$ is unsorted

1. $\boldsymbol{s} = \emptyset$ and $\boldsymbol{u} = \boldsymbol{a}$
2. Remove the first elements x of $\boldsymbol{u}$
3. Insert x into its proper place among $\boldsymbol{s}$

Repeat the actions 2-3 until $\boldsymbol{u}$ is empty

Implementation

- Algorithm 1

```
template <class Item>
void insertion(Item a[], int l, int r) {
    Item v;
    for (int i = l + 1; i <= r; i++) {
        v = a[i];
        for (int j = i; j > l && v < a[j - 1]; j--)
            a[j] = a[j - 1];
        a[j] = v;
    }
}
```

Implementation (cont.)

- Algorithm 2 (using *sentinel technique*)

```
template <class Item>
void insertion(Item a[], int l, int r) {
  int i;
  for (i = r; i > l; i--) compare_swap(a[i - 1], a[i]);
  for (i = l + 2; i <= r; i++) {
    v = a[i];
    for (int j = i; v < a[j - 1]; j--)
      a[j] = a[j - 1];
    a[j] = v;
  }
}
```

Illustration

- Trace (right after the inner loop is exhausted)

a[]

i	j	0	1	2	3	4	5	6	7	8	9	10
		S	O	R	T	E	X	A	M	P	L	E
1	0	O	S	R	T	E	X	A	M	P	L	E
2	1	O	R	S	T	E	X	A	M	P	L	E
3	3	O	R	S	T	E	X	A	M	P	L	E
4	0	E	O	R	S	T	X	A	M	P	L	E
5	5	E	O	R	S	T	X	A	M	P	L	E
6	0	A	E	O	R	S	T	X	M	P	L	E
7	2	A	E	M	O	R	S	T	X	P	L	E
8	4	A	E	M	O	P	R	S	T	X	L	E
9	2	A	E	L	M	O	P	R	S	T	X	E
10	2	A	E	E	L	M	O	P	R	S	T	X
		A	E	E	L	M	O	P	R	S	T	X

Analysis

Theorem 2

The number of exchanges used by insertion sort is equal to the number of inversions in the array, and the number of compares is at least equal to the number of inversions and at most equal to the number of inversions plus the array size minus 1.

Theorem 3

Insertion sort uses $\sim N^2/4$ compares and $\sim N^2/4$ exchanges to sort a randomly ordered array of length N with distinct keys, on the average.

Analysis (cont.)

- Time complexity:

best case	?
average case	?
worst case	?

- Space complexity: $O(1)$
- Stability: Yes

Bubble Sort

Bubble Sort

- Keep passing through the array, exchanging adjacent elements that are out of order, continuing until the array is sorted.
  For example, in the array

1	3	8	2	9	2	5	6

  the first round of bubble sort swaps elements as follows:

1	3	2	8	9	2	5	6

1	3	2	8	2	9	5	6

1	3	2	8	2	5	9	6

1	3	2	8	2	5	6	9

- Bubble Sort is a kind of Selection Sort.

Bubble Sort

```
template <class Item>
void bubble(Item a[], int l, int r) {
  for (int i = l; i < r; i++)
    for (int j = r; j > i; j--)
      compare_swap(a[j - 1], a[j]);
}
```

- **Challenge**: reimplement the function `bubble` using recursion technique

Analysis

- Time complexity:

best case	?
average case	?
worst case	?

- Space complexity: $O(1)$
- Stability: Yes

Shell Sort

Shell Sort

- Insertion sort is slow because the only exchanges it does involve adjacent items, so items can move through the array only one place at a time.
- Shellsort is a simple extension of insertion sort that gains speed by allowing exchanges of elements that are far apart.
- The running time is better than $O(n^2)$

Concept 7

An h-sorted array is h independent sorted subsequences, interleaved together.

h = 4

L	E	E	A	M	H	L	E	P	S	O	L	T	S	X	R
L				M				P				T			
	E				H				S				S		
		E				L				O				X	
			A				E				L				R

Shell Sort

Idea

- Given the decrement sequence $\{h_1, h_2, ..., h_t\}$ where $h_i \in \mathbb{N}$ and $h_t = 1$
- For each $h \in \{h_1, h_2, ..., h_t\}$
  1. Split an array $\boldsymbol{a}$ into h subsequences

$$
\begin{array}{l}
a_0, a_{0+h}, a_{0+2h}, \cdots \\
a_1, a_{1+h}, a_{1+2h}, \cdots \\
a_2, a_{2+h}, a_{2+2h}, \cdots \\
\cdots
\end{array}
$$

  2. Using *Insertion Sort* to sort each subsequence

Increment/Decrement Sequence

- Shell proposed

$$\begin{aligned} h_1 &= \frac{N}{2} \\ h_{i+1} &= \frac{h_i}{2} \quad i > 1 \end{aligned} \tag{1}$$

- Hibbard proposed

$$h_i = 2^i - 1 \tag{2}$$

- Knuth proposed

$$\begin{aligned} h_1 &= 1 \\ h_{i+1} &= 3h_i + 1 \quad i > 1 \end{aligned} \tag{3}$$

- Pratt proposed

$$\text{Successive numbers of the form } 2^p 3^q, \quad p, q \in \mathbb{N} \tag{4}$$

Implementation

```
template <class Item>
void shellsort(Item a[], int l, int r) {
  int h;
  for (h = 1; h <= (r - l) / 9; h = 3 * h + 1);
  for (; h > 0; h /= 3)
    for (int i = l + h; i <= r; i++) {
      int j = i;
      Item v = a[i];
      while (j >= l + h && v < a[j - h]) {
        a[j] = a[j - h];
        j -= h;
      }
      a[j] = v;
    }
}
```

Illustration

input	S	H	E	L	L	S	O	R	T	E	X	A	M	P	L	E
13-sort	P	H	E	L	L	S	O	R	T	E	X	A	M	S	L	E
	P	H	E	L	L	S	O	R	T	E	X	A	M	S	L	E
	P	H	E	L	L	S	O	R	T	E	X	A	M	S	L	E
4-sort	L	H	E	L	P	S	O	R	T	E	X	A	M	S	L	E
	L	H	E	L	P	S	O	R	T	E	X	A	M	S	L	E
	L	H	E	L	P	S	O	R	T	E	X	A	M	S	L	E
	L	H	E	L	P	S	O	R	T	E	X	A	M	S	L	E
	L	H	E	L	P	S	O	R	T	E	X	A	M	S	L	E
	L	E	E	L	P	H	O	R	T	S	X	A	M	S	L	E
	L	E	E	L	P	H	O	R	T	S	X	A	M	S	L	E
	L	E	E	A	P	H	O	L	T	S	X	R	M	S	L	E
	L	E	E	A	M	H	O	L	P	S	X	R	T	S	L	E
	L	E	E	A	M	H	O	L	P	S	X	R	T	S	L	E
	L	E	E	A	M	H	L	L	P	S	O	R	T	S	X	E
	L	E	E	A	M	H	L	E	P	S	O	L	T	S	X	R
1-sort	E	L	E	A	M	H	L	E	P	S	O	L	T	S	X	R
	E	E	L	A	M	H	L	E	P	S	O	L	T	S	X	R
	A	E	E	L	M	H	L	E	P	S	O	L	T	S	X	R
	A	E	E	L	M	H	L	E	P	S	O	L	T	S	X	R
	A	E	E	H	L	M	L	E	P	S	O	L	T	S	X	R
	A	E	E	H	L	L	M	E	P	S	O	L	T	S	X	R
	A	E	E	E	H	L	L	M	P	S	O	L	T	S	X	R
	A	E	E	E	H	L	L	M	P	S	O	L	T	S	X	R
	A	E	E	E	H	L	L	M	P	S	O	L	T	S	X	R
	A	E	E	E	H	L	L	M	O	P	S	L	T	S	X	R
	A	E	E	E	H	L	L	L	M	O	P	S	T	S	X	R
	A	E	E	E	H	L	L	L	M	O	P	S	T	S	X	R
	A	E	E	E	H	L	L	L	M	O	P	S	S	T	X	R
	A	E	E	E	H	L	L	L	M	O	P	S	S	T	X	R
	A	E	E	E	H	L	L	L	M	O	P	R	S	S	T	X
result	A	E	E	E	H	L	L	L	M	O	P	R	S	S	T	X

Analysis

Theorem 4

The result of h-sorting an array that is k-ordered is an array that is both h- and k-ordered

Theorem 5

*Shellsort does less than $N(h-1)(k-1)/g$ **comparisons** to g-sort an array that is h- and k-ordered, provided that h and k are relatively prime*

Analysis (cont.)

- Time complexity:

best case	?
average case	?
worst case	?

- Space complexity: $O(1)$
- Stability: No

Heap Sort

Heap Sort

Concept 8

- **Max heap**: A tree is heap-ordered if the key in each node is larger than or equal to the keys in all of that node's children (if any)
- **Min heap**: A tree is heap-ordered if the key in each node is smaller than or equal to the keys in all of that node's children (if any)

Theorem 6

- *Max heap: No node in a heap-ordered tree has a key larger than the key at the root*
- *Min heap: No node in a heap-ordered tree has a key smaller than the key at the root*

Heap Representation

- Array representation of a heap-ordered complete binary tree

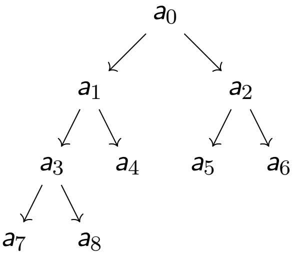


Figure 1: Array $\{a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8\}$ and complete binary tree

Heap Representation (cont.)

Note

Array vs. Complete binary tree

- Root node is a_0
- $\textsc{Parent}(a_i)$ is $a_{\left\lfloor \frac{i-1}{2} \right\rfloor}$ or nothing
- $\textsc{LeftChild}(a_i)$ is a_{2i+1} or nothing
- $\textsc{RightChild}(a_i)$ is a_{2i+2} or nothing

Top-down heapify

At the given node a_i

- **Exchange** the key in the given node a_i with the largest key among that node's children a_{2i+1} and a_{2i+2}
- **Move down** to that child, and continuing down the tree until we reach the bottom or a point where no child has a larger key.

Implementation

```
template <class Item>
void heapify(Item a[], int n, int i) {
  Item v = a[i];
  while (i < n / 2) {
    int child = 2 * i + 1;
    if (child < n - 1)
      if (a[child] > a[child + 1])
        child++;
    if (v >= a[child]) break;
    a[i] = a[child];
    i = child;
  }
  a[i] = v;
}
```

Heap Sort

- Build max-heap array: use `heapify` operation to convert an array $\boldsymbol{a}$ to a max-heap array
  - All elements in the range $\left\{a_{\frac{n}{2}}, \ldots, a_{n-1}\right\}$ are leaf nodes.
  - Apply `heapify` operation for these elements $\left\{a_{\frac{n}{2}-1}, \ldots, a_0\right\}$
- Sort a max-heap array $\boldsymbol{a}$
  1. Swap the first and the last element
  2. Remove the last element

  If $|\boldsymbol{a}| > 1$ then apply `heapify` operation for a_0 and repeat actions 1-2

Heap Sort (cont.)

```
template <class Item>
void heapsort(Item a[], int l, int r) {
  Item *pa = a + l;
  int N = r - l + 1;
  for (int k = N / 2 - 1; k >= 0; k--)
    heapify(pa, N, k);
  while (N > 1) {
    swap(pa[0], pa[N - 1]);
    N--;
    heapify(pa, N, 0);
  }
}
```

Analysis

Theorem 7

Heapsort uses fewer than $2N \log_2 N$ comparisons to sort N elements

- Time complexity:

best case	?
average case	?
worst case	?

- Space complexity:
- Stability:

Merge Sort

Top-Down Merge Sort

Idea

Merge sort sorts a subarray a$[l \dots r]$ as follows:

1. If $l \geq r$, do not do anything, because the subarray is already sorted or empty.
2. Calculate the position of the middle element: $m = \lfloor (l + r)/2 \rfloor$.
3. Recursively sort the subarray a$[l \dots m]$.
4. Recursively sort the subarray a$[m + 1 \dots r]$.
5. *Merge* the sorted subarrays a$[l \dots m]$ and a$[m + 1 \dots r]$ into a sorted subarray a$[l \dots r]$.

Illustration

Sorting the following array:

1	3	6	2	8	2	5	9

- The array will be divided into two subarrays as follows:

1	3	6	2

8	2	5	9

- Then, the subarrays will be sorted recursively as follows:

1	2	3	6

2	5	8	9

- Finally, the algorithm merges the sorted subarrays and creates the final sorted array:

1	2	2	3	5	6	8	9

Implementation

```
template <class Item>
void mergesort(Item a[], Item aux[], int l, int r) {
  if (r <= l) return;
  int m = (l + r) / 2;
  mergesort(a, aux, l, m);
  mergesort(a, aux, m + 1, r);
  merge(a, aux, l, m, r);
}
```

Implementation (cont.)

```
template <class Item>
void merge(Item a[], Item aux[], int l, int m, int r) {
  int i, j, k;
  for (k = l; k <= r; k++)
    aux[k] = a[k];
  i = l;  j = m + 1;  k = l;
  while (i <= m && j <= r)
    if (aux[i] <= aux[j]) a[k++] = aux[i++];
    else a[k++] = aux[j++];
  while (i <= m)
    a[k++] = aux[i++];
  while (j <= r)
    a[k++] = aux[j++];
}
```

Bottom-up Merge Sort

Idea

Bottom-up merge sort consists of

- A sequence of passes over the whole array doing `sz`-by-`sz` merges
- Doubling `sz` on each pass.
- The final subarray is of size `sz` only if the array size is an even multiple of `sz`, so the final merge is an `sz`–by-`x` merge, for some `x` less than or equal to `sz`.

Implementation

```
template <class Item>
void mergesort(Item a[], Item aux[], int l, int r) {
  for (int sz = 1; sz <= r - l; sz = sz + sz)
    for (int i = l; i <= r - sz; i += sz + sz)
      merge(a, aux, i, i + sz - 1, min(i + sz + sz - 1, r));
}
```

Analysis

Theorem 8

Mergesort requires about $N \log_2 N$ comparisons to sort any array of N elements

- Time complexity:

best case	?
average case	?
worst case	?

- Space complexity:
- Stability:

Quick Sort

Quick Sort

Idea

Quicksort invented by C. A. R. Hoare in 1960 is a divide-and-conquer method for sorting

1. Partition an array $\boldsymbol{a}$ into two parts $\boldsymbol{a}_{left}$ and $\boldsymbol{a}_{right}$ such that $\forall x \in \boldsymbol{a}_{left}$ and $\forall y \in \boldsymbol{a}_{right}$ then $x \leq y$
2. Sort the parts $\boldsymbol{a}_{left}$ and $\boldsymbol{a}_{right}$ independently
3. Join $\boldsymbol{a}_{left}$ and $\boldsymbol{a}_{right}$ to $\boldsymbol{a} = \boldsymbol{a}_{left}\boldsymbol{a}_{right}$

Implementation

```cpp
template <class Item>
void quicksort(Item a[], int l, int r) {
  if (r <= l) return;
  int i = partition(a, l, r);
  quicksort(a, l, i - 1);
  quicksort(a, i + 1, r);
}
```

Implementation (cont.)

```
template <class Item>
int partition(Item a[], int l, int r) {
  int i = l - 1, j = r; Item v = a[r];
  for (;;) {
    while (a[++i] < v);
    while (v < a[--j]) if (j == l) break;
    if (i >= j) break;
    swap(a[i], a[j]);
  }
  swap(a[i], a[r]);
  return i;
}
```

Analysis

Theorem 9

Quicksort uses $\sim 2N \log_2 N$ ***compares*** *(and one-sixth that many exchanges) on the average to sort an array of length N with distinct keys.*

Theorem 10

Quicksort uses $\sim N^2/2$ ***compares*** *in the worst case*

Analysis (cont.)

- Time complexity:

best case	?
average case	?
worst case	?

- Space complexity: $O(1)$?
- Stability: No

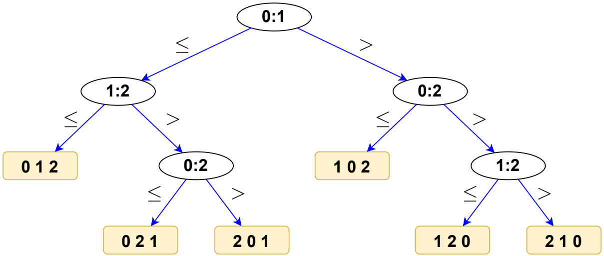
A lower bound for the worst case

Theorem 11

No compare-based sorting algorithm can guarantee to sort N items with fewer than $\log_2(N!) \sim N \log_2 N$ ***compares****.*

Proof

Comparison sorts can be viewed abstractly in terms of **decision trees**.



■

Comparing Sorting Algorithms

- Performance characteristics of sorting algorithms

algorithm	stable?	in-place?	running time	space
selection	no	yes	N^2	1
insertion	yes	yes	between N and N^2	1
shell	no	yes	$N \log N$?, $N^{6/5}$?	1
merge	yes	no	$N \log N$	N
quick	no	yes	$N \log N$	$\log N$
3-way quick	no	yes	between N and $N \log N$	$\log N$
heap	no	yes	$N \log N$	1

Key-indexed Counting

Assumptions about keys

Assumption. Keys are integers between 0 and $R-1$.
Implication. Can use key as an array index.

input *name*	*section*	sorted result (*by section*)	
Anderson	2	Harris	1
Brown	3	Martin	1
Davis	3	Moore	1
Garcia	4	Anderson	2
Harris	1	Martinez	2
Jackson	3	Miller	2
Johnson	4	Robinson	2
Jones	3	White	2
Martin	1	Brown	3
Martinez	2	Davis	3
Miller	2	Jackson	3
Moore	1	Jones	3
Robinson	2	Taylor	3
Smith	4	Williams	3
Taylor	3	Garcia	4
Thomas	4	Johnson	4
Thompson	4	Smith	4
White	2	Thomas	4
Williams	3	Thompson	4
Wilson	4	Wilson	4

keys are small integers

Key-indexed Counting

Problem. Sort an array a of N items whose keys are integers between 0 and $R-1$.

```
// Compute frequency counts.
for (int i = 0; i < N; i++)
  count[a[i].key() + 1]++;
// Transform counts to indices.
for (int r = 0; r < R; r++)
  count[r + 1] += count[r];
// Distribute the items.
for (int i = 0; i < N; i++)
  aux[count[a[i].key()]++] = a[i];
// Copy back.
for (int i = 0; i < N; i++)
  a[i] = aux[i];
```

Illustration

- Input an array a of 12 letters

i	a[i]
0	d
1	a
2	c
3	f
4	f
5	b
6	d
7	b
8	f
9	b
10	e
11	a

use a for 0
b for 1
c for 2
d for 3
e for 4
f for 5

Illustration (cont.)

- Compute frequency counts

i	a[i]
0	d
1	a
2	c
3	f
4	f
5	b
6	d
7	b
8	f
9	b
10	e
11	a

offset by 1
[stay tuned]

r	count[r]
a	0
b	2
c	3
d	1
e	2
f	1
-	3

Illustration (cont.)

- Transform counts to indices.

i	a[i]
0	d
1	a
2	c
3	f
4	f
5	b
6	d
7	b
8	f
9	b
10	e
11	a

r	count[r]
a	0
b	2
c	5
d	6
e	8
f	9
-	12

Illustration (cont.)

- Distribute the data

i	a[i]
0	d
1	a
2	c
3	f
4	f
5	b
6	d
7	b
8	f
9	b
10	e
11	a

r	count[r]
a	2
b	5
c	6
d	8
e	9
f	12
-	12

i	aux[i]
0	a
1	a
2	b
3	b
4	b
5	c
6	d
7	d
8	e
9	f
10	f
11	f

Illustration (cont.)

- Copy back

i	a[i]
0	a
1	a
2	b
3	b
4	b
5	c
6	d
7	d
8	e
9	f
10	f
11	f

r	count[r]
a	2
b	5
c	6
d	8
e	9
f	12
-	12

i	aux[i]
0	a
1	a
2	b
3	b
4	b
5	c
6	d
7	d
8	e
9	f
10	f
11	f

Analysis

Theorem 12

Key-indexed counting uses $8N + 3R + 1$ array accesses to stably sort N items whose keys are integers between 0 and $R - 1$.

- Time complexity:

best case	?
average case	?
worst case	?

- Space complexity:
- Stability:

Radix Sort

Radix Sort

Concept 9

- A **byte** is a fixed-length sequence of bits.
- A **word** is a fixed-length sequence of bytes.
- A **string** is a variable-length sequence of bytes.

Radix Sort (cont.)

Idea

Radix-sorting algorithms treat the **keys** as numbers represented in a base-R number system, for various values of R (the radix), and work with individual **digits** of the numbers.

Concept 10

A key is a radix-R number, with digits numbered from the right (starting at 0)

- In programing, we use the abstract `digit` operation to access digits of keys

Radix Sort (cont.)

There are two basic approaches to radix sorting:

- The first class of methods: They examine the digits in the keys in a left-to-right order, working with the most significant digits first. These methods are generally referred to as **most-significant-digit** (MSD) radix sorts or **top-down methods**.
- The second class of methods: They examine the digits in the keys in a right-to-left order, working with the least significant digits first. These methods are generally referred to as **least-significant-digit** (LSD) radix sorts or **bottom-up methods**.

LSD Implementation

```
template <class Item>
void radixsort(Item a[], int l, int r) {
    vector<Item> bins[radix];
    for (int d = 0; d < max_digit; d++) {
        // clear bins
        ...
        // distribute
        for (i = l; i <= r; i++)
            bins[digit(a[i],d,radix)].push_back(a[i]);
        // join bins to a[l...r]
        ...
    }
}
```

Example 1

Problem Sort an array of integer numbers `{170, 45, 75, 90, 802, 2, 24, 66}`

- Rewrite the numbers in 3-digit format `{170, 045, 075, 090, 802, 002, 024, 066}`

Example 1 (cont.)

- Distribute the data into 10 bins on digit $d = 0$

{170, 045, 075, 090, 802, 002, 024, 066}

```
bin 0: 170 090
bin 1:
bin 2: 802 002
bin 3:
bin 4: 024
bin 5: 045 075
bin 6: 066
bin 7:
bin 8:
bin 9:
```

- Join the bins

{170, 090, 802, 002, 024, 045, 075, 066}

Example 1 (cont.)

- Distribute the data into 10 bins on digit $d = 1$

{170, 090, 802, 002, 024, 045, 075, 066}

```
bin 0: 802, 002
bin 1:
bin 2: 024
bin 3:
bin 4: 045
bin 5:
bin 6: 066
bin 7: 170, 075
bin 8:
bin 9: 090
```

- Join the bins

{802, 002, 024, 045, 066, 170, 075, 090}

Example 1 (cont.)

- Distribute the data into 10 bins on digit $d = 2$

  {802, 002, 024, 045, 066, 170, 075, 090}

  bin 0: 002, 024, 045, 066, 075, 090
  bin 1: 170
  bin 2:
  bin 3:
  bin 4:
  bin 5:
  bin 6:
  bin 7:
  bin 8: 802
  bin 9:

- Join the bins

  {002, 024, 045, 066, 075, 090, 170, 802}

Example 1 (cont.)

170		170		802		002
045		090		002		024
075		802		024		045
090	$\longrightarrow$	002	$\longrightarrow$	045	$\longrightarrow$	066
802		024		066		075
002		045		170		090
024		075		075		170
066		066		090		802

Example 2

Problem Sort 17 integer numbers

{6, 7, 1, 3, 5, 2, 0, 4, 2, 1, 7, 2, 1, 3, 5, 2, 7}

- Rewrite the numbers in 3-digit binary numbers

110	111	001	011	101	010	000	100	010	001	111	010	001	011	101	010	111

Example 2 (cont.)

110	111	001	011	101	010	000	100	010	001	111	010	001	011	101	010	111

$\downarrow$

$\downarrow$

$\downarrow$

Analysis

- Time complexity:

best case	?
average case	?
worst case	?

- Space complexity:
- Stability:

Workshop

Quiz

1. What is a sorting operation?

2. What is an inversion?

3. How selection sort sorts the sample array `E A S Y Q U E S T I O N`?

Exercises

- Programming exercises in [Cormen, 2009, Sedgewick, 2002]

References

Cormen, T. H. (2009).
Introduction to algorithms.
MIT press.

Sedgewick, R. (2002).
Algorithms in Java, Parts 1-4, volume 1.
Addison-Wesley Professional.

Walls and Mirrors (2014).
Data Abstraction And Problem Solving with C++.
Pearson.