CS 240 – Data Structures and Data Management

Module 3: Sorting and Randomized Algorithms

A. Storjohann

Based on lecture notes by many previous cs240 instructors

David R. Cheriton School of Computer Science, University of Waterloo

Fall 2018

References: Sedgewick 6.10, 7.1, 7.2, 7.8, 10.3, 10.5
Goodrich & Tamassia 4.4

version 2018–09–24 10:11

Outline

1. Sorting and Randomized Algorithms
    - QuickSelect
    - Randomized Algorithms
    - QuickSort
    - Lower Bound for Comparison-Based Sorting
    - Non-Comparison-Based Sorting

Outline

1. Sorting and Randomized Algorithms
   - QuickSelect
   - Randomized Algorithms
   - QuickSort
   - Lower Bound for Comparison-Based Sorting
   - Non-Comparison-Based Sorting

Selection vs. Sorting

The *selection problem*: Given an array A of n numbers, and $0 \leq k < n$, find the element that would be at position k of the sorted array.

0	1	2	3	4	5	6	7	8	9
30	60	10	0	50	80	90	20	40	70

select(3) should return 30.

Best heap-based algorithm had running time $\Theta(n + k \log n)$.
For *median finding* (selection with $k = \lfloor \frac{n}{2} \rfloor$) this is $\Theta(n \log n)$.

This is the same cost as our best sorting algorithms.

Question: Can we do selection in linear time?
The *quick-select* algorithm answers this question in the affirmative.

The encountered sub-routines will also be useful for other sorting algorithms.

Crucial Subroutines

quick-select and the related algorithm *quick-sort* rely on two subroutines:

- *choose-pivot*(A): Choose an index p. We will use the *pivot-value* $v \leftarrow A[p]$ to rearrange the array.
- *partition*(A, p): Rearrange A and return *pivot-index* i so that
    - the pivot-value v is in $A[i]$,
    - all items in $A[0, \dots, i-1]$ are $\leq v$, and
    - all items in $A[i+1, \dots, n-1]$ are $\geq v$.

Simplest idea for *choose-pivot*: Always select rightmost element in array

```
choose-pivot1(A)
1.      return A.size()-1
```

We will consider more sophisticated ideas later on.

Partition Algorithm

Conceptually easy linear-time implementation:

```
$partition(A, p)$
$A$: array of size $n$,   $p$: integer s.t. $0 \le p < n$
    Create empty lists $small$ and $large$.
    $v \leftarrow A[p]$
    **for** each element $x$ in $A[0, \dots, p-1]$ or $A[p+1 \dots n-1]$
        **if** $x < v$ append $x$ to $small$
        **else** append $x$ to $large$
    $i \leftarrow size(small)$
    Overwrite $A[0 \dots i-1]$ by elements in $small$
    Overwrite $A[i]$ by $v$
    Overwrite $A[i+1 \dots n-1]$ by elements in $large$
    return $i$
```

More challenging: partition *in place*

Efficient In-Place partition (Hoare)

i=-1	0	1	2	3	4	5	6	7	8	j=9
	30	60	10	0	50	80	90	20	40	v=70

0	1	2	3	4	i=5	6	7	j=8	9
30	60	10	0	50	80	90	20	40	v=70

0	1	2	3	4	i=5	6	7	j=8	9
30	60	10	0	50	40	90	20	80	v=70

0	1	2	3	4	5	i=6	j=7	8	9
30	60	10	0	50	40	90	20	80	v=70

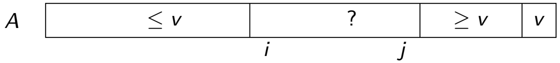
0	1	2	3	4	5	i=6	j=7	8	9
30	60	10	0	50	40	20	90	80	v=70

0	1	2	3	4	5	j=6	i=7	8	9
30	60	10	0	50	40	20	90	80	v=70

0	1	2	3	4	5	j=6	i=7	8	9
30	60	10	0	50	40	20	70	80	90

Efficient In-Place partition (Hoare)

Idea: Keep swapping the outer-most wrongly-positioned pairs.



```
partition($A, p$)
$A$: array of size $n$,   $p$: integer s.t. $0 \leq p < n$
 1.   swap($A[n-1], A[p]$)
 2.   $i \leftarrow -1$,   $j \leftarrow n-1$,   $v \leftarrow A[n-1]$
 3.   **loop**
 4.        **do** $i \leftarrow i+1$ **while** $i < n$ and $A[i] < v$
 5.        **do** $j \leftarrow j-1$ **while** $j > 0$ and $A[j] > v$
 6.        **if** $i \geq j$ **then break**   (goto 9)
 7.        **else** swap($A[i], A[j]$)
 8.   **end loop**
 9.   swap($A[n-1], A[i]$)
10.   **return** $i$
```

Running time: $\Theta(n)$.

QuickSelect Algorithm

```
quick-select1(A, k)
A: array of size n,   k: integer s.t. $0 \leq k < n$
1.   $p \leftarrow$ choose-pivot1(A)
2.   $i \leftarrow$ partition(A, p)
3.   if $i = k$ then
4.       return $A[i]$
5.   else if $i > k$ then
6.       return quick-select1($A[0, 1, \ldots, i-1], k$)
7.   else if $i < k$ then
8.       return quick-select1($A[i+1, i+2, \ldots, n-1], k-i-1$)
```

Analysis of *quick-select1*

Worst-case analysis: Recursive call could always have size $n-1$.
Recurrence given by

$$T(n) = \begin{cases} T(n-1) + cn, & n \geq 2 \\ c, & n = 1 \end{cases}$$

for some constant $c > 0$.

Solution: $T(n) = cn + c(n-1) + c(n-2) + \cdots + c \cdot 2 + c \in \Theta(n^2)$

Best-case analysis: First chosen pivot could be the kth element
No recursive calls; total cost is $\Theta(n)$.

Sorting Permutations

- Need to take average running time over all inputs.
- How to characterize input of size *n*?
  (There are infinitely many sets of *n* numbers.)
- Simplifying assumption: All input numbers are *distinct*.
- Observe: quick-select1 would act the same on inputs
  14, 2, 3, 6, 1, 11, 7 and
  14, 2, 4, 6, 1, 12, 8
- The actual numbers do not matter, only their *relative order*.
- Characterize input via *sorting permutation*: the permutation that would put the input in order.
- Assume all *n*! permutations are *equally likely*.

$\rightsquigarrow$ Average cost is sum of costs for all permutations, divided by *n*!

Average-Case Analysis of *quick-select1*

- Define $T(n)$ to be the average cost for selecting from size-n array, presuming we use *choose-pivot1*(A).
- Fix one $0 \leq i \leq n-1$. There are $(n-1)!$ permutations for which the pivot-value v is the ith smallest item, i.e., the pivot-index is i.

$$\begin{aligned} T(n) &= \frac{1}{n!} \sum_{I:\text{size}(I)=n} \text{running time for instance } I \\ &\leq c \cdot n + \frac{1}{n} \sum_{i=0}^{n-1} \max\{T(i), T(n-i-1)\} \end{aligned}$$

Average-Case Analysis of *quick-select1*

$$T(n) \leq c \cdot n + \frac{1}{n} \sum_{i=0}^{n-1} \max\{T(i), T(n-i-1)\}$$

Theorem: $T(n) \in \Theta(n)$.
Proof:

Outline

1. Sorting and Randomized Algorithms
   - QuickSelect
   - Randomized Algorithms
   - QuickSort
   - Lower Bound for Comparison-Based Sorting
   - Non-Comparison-Based Sorting

Randomized algorithms

A *randomized algorithm* is one which relies on some random numbers in addition to the input.
The cost will depend on the input and the random numbers used.

Goal: *no more bad instances, just unlucky numbers.*
$\rightarrow$ Shift the dependency of costs from what we can't control (the input), to what we can control (the random numbers).

Expected running time

Define $T(I,R)$ to be the running time of the randomized algorithm for an instance I and the sequence of random numbers R.

The *expected running time* $T^{(\mathrm{exp})}(I)$ for instance I is the expected value for $T(I,R)$:

$$T^{(\mathrm{exp})}(I) = \mathbf{E}[T(I,R)] = \sum_{R} T(I,R) \cdot \Pr[R]$$

The *worst-case expected running time* is

$$T^{(\mathrm{exp})}_{\mathrm{worst}}(n) = \max_{\{I \,:\, size(I)=n\}} T^{(\mathrm{exp})}(I).$$

The *average-case expected running time* is

$$T^{(\mathrm{exp})}_{\mathrm{avg}}(n) = \frac{1}{|\{I : size(I) = n\}|} \sum_{\{I : size(I)=n\}} T^{(\mathrm{exp})}(I).$$

For well-designed randomized algorithms, these are the same (why?).

Randomized QuickSelect: Shuffle

random(n) returns an integer uniformly from $\{0, 1, 2, \ldots, n-1\}$.

First idea: Randomly permute the input first using *shuffle*:

```
shuffle(A)
A: array of size n
1.    for $i \leftarrow 0$ to $n-2$ do
2.        swap( $A[i], A[i + random(n-i)]$ )
```

Expected cost becomes the same as the average cost: $\Theta(n)$.

Randomized QuickSelect: Random Pivot

Second idea: Change the pivot selection.

```
choose-pivot2(A)
1.    return random(n)
```

```
quick-select2(A, k)
1.    p ← choose-pivot2(A)
2.    ...
```

With probablity $\frac{1}{n}$ the random pivot has index i, so the analysis is just like that for the average-case. The expected running time is again $\Theta(n)$.

This is generally the fastest quick-select implementation.

There exists a variation that has *worst-case* running time $O(n)$, but it uses double recursion and is slower in practice. ($\rightsquigarrow$ *cs*341)

Outline

1. Sorting and Randomized Algorithms
   - QuickSelect
   - Randomized Algorithms
   - QuickSort
   - Lower Bound for Comparison-Based Sorting
   - Non-Comparison-Based Sorting

QuickSort

Hoare developed *partition* and *quick-select* in 1960; together with a *sorting* method based on partitioning:

```
quick-sort1(A)
A: array of size n
1.    if $n \leq 1$ then return
2.    $p \leftarrow$ choose-pivot1(A)
3.    $i \leftarrow$ partition(A, p)
4.    quick-sort1($A[0, 1, \ldots, i-1]$)
5.    quick-sort1($A[i+1, \ldots, n-1]$)
```

Worst case: $T^{(\text{worst})}(n) = T^{(\text{worst})}(n-1) + \Theta(n)$
Same as *quick-select1*: $T^{(\text{worst})}(n) \in \Theta(n^2)$

Best case: $T^{(\text{best})}(n) = T^{(\text{best})}(\lfloor \frac{n-1}{2} \rfloor) + T^{(\text{best})}(\lceil \frac{n-1}{2} \rceil) + \Theta(n)$
Similar to *merge-sort*: $T^{(\text{best})}(n) \in \Theta(n \log n)$

Average-case analysis of quick-sort1

- As before, $\frac{1}{n}$ of permutations have pivot-index i.
- The recursive work is then $T^{(\mathrm{avg})}(i) + T^{(\mathrm{avg})}(n-i-1)$.
- So average running time is

$$T^{(\mathrm{avg})}(n) = c \cdot n + \frac{1}{n}\sum_{i=0}^{n-1}\big(T^{(\mathrm{avg})}(i) + T^{(\mathrm{avg})}(n-i-1)\big), \qquad n \geq 2$$

- **Theorem:** $T^{(\mathrm{avg})}(n) \in \Theta(n \log n)$.
  Proof:

More notes on QuickSort

- We can randomize by using *choose-pivot2*, giving $\Theta(n \log n)$ *expected time* for *quick-sort2*.
- The auxiliary space is O(recursion depth).
  - This is $\Theta(n)$ in the worst-case.
  - It can be reduced to $\Theta(\log n)$ worst-case by recursing in smaller sub-array first and replacing the other recursion by a while-loop.
- One should stop recursing when $n \leq 10$.
  One run of InsertionSort at the end then sorts everything in $O(n)$ time since all items are within 10 units of their required position.
- Arrays with many duplicates can be sorted faster by changing *partition* to produce three subsets

$\leq v$	$= v$	$\geq v$

- QuickSort is often the most efficient algorithm in practice.

Outline

1. Sorting and Randomized Algorithms
   - QuickSelect
   - Randomized Algorithms
   - QuickSort
   - Lower Bound for Comparison-Based Sorting
   - Non-Comparison-Based Sorting

Lower bounds for sorting

We have seen many sorting algorithms:

Sort	Running time	Analysis
Selection Sort	$\Theta(n^2)$	worst-case
Insertion Sort	$\Theta(n^2)$	worst-case
Merge Sort	$\Theta(n \log n)$	worst-case
Heap Sort	$\Theta(n \log n)$	worst-case
quick-sort1	$\Theta(n \log n)$	average-case
quick-sort2	$\Theta(n \log n)$	expected
quick-sort3	$\Theta(n \log n)$	worst-case

Question: Can one do better than $\Theta(n \log n)$ running time?

Answer: Yes and no! *It depends on what we allow*.

- No: Comparison-based sorting lower bound is $\Omega(n \log n)$.
- Yes: Non-comparison-based sorting can achieve $O(n)$ (under restrictions!). $\rightarrow$ see below

The Comparison Model

In the *comparison model* data can only be accessed in two ways:

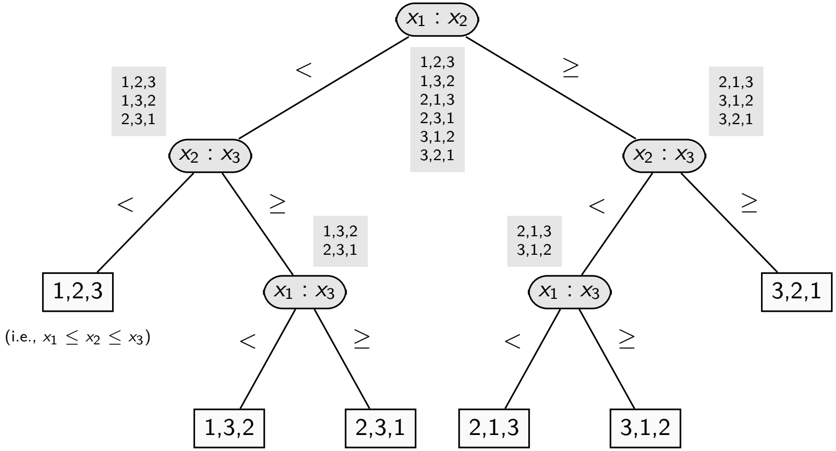
- comparing two elements
- moving elements around (e.g. copying, swapping)

This makes very few assumptions on the kind of things we are sorting.
We count the number of above operations.

All sorting algorithms seen so far are in the comparison model.

Decision trees

Comparison-based algorithms can be expressed as *decision tree*.
To sort $\{x_1, x_2, x_3\}$:



Lower bound for sorting in the comparison model

Theorem. Any correct **comparison-based** sorting algorithm requires at least $\Omega(n \log n)$ comparison operations.

Proof.

Outline

1. Sorting and Randomized Algorithms
    - QuickSelect
    - Randomized Algorithms
    - QuickSort
    - Lower Bound for Comparison-Based Sorting
    - Non-Comparison-Based Sorting

Non-Comparison-Based Sorting

- Assume keys are numbers in base R (R: radix)
  - $R = 2, 10, 128, 256$ are the most common.

Example ($R = 4$):

123	230	21	320	210	232	101

- Assume all keys have the same number m of digits.
  - Can achieve after padding with leading 0s.

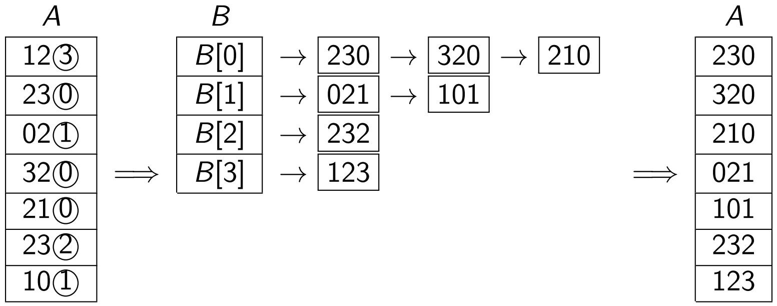
Example ($R = 4$):

123	230	021	320	210	232	101

- Can sort based on individual digits.
  - How to sort 1-digit numbers?
  - How to sort multi-digit numbers based on this?

Bucket Sort

Sort array *A* by last digit:



Bucket Sort

- Sorts numbers by a single digit.
- Create a "bucket" for each possible digit: Array $B[0..R-1]$ of lists
- Copy item with digit i into bucket $B[i]$
- At the end copy buckets in order into A.

```
Bucket-sort(A, d)
A: array of size n, contains numbers with digits in {0, ..., R − 1}
d: index of digit by which we wish to sort
1.    Initialize an array B[0...R − 1] of empty lists
2.    for i ← 0 to n − 1 do
3.        Append A[i] at end of B[d^th digit of A[i]]
4.    i ← 0
5.    for j ← 0 to R − 1 do
6.        while B[j] is non-empty do
7.            move first element of B[j] to A[i++]
```

- This is *stable*: equal items stay in original order.
- Run-time $\Theta(n + R)$, auxiliary space $\Theta(n)$

Count Sort

- Bucket sort wastes space for linked lists.
- Observe: We know exactly where numbers in $B[j]$ go:
  - The first of them is at index $|B[0]| + |B[1]| + \cdots + |B[j-1]|$
  - The others follow.
- So we don't need the lists; it's enough to count how many there would be in it.

Count Sort Pseudocode

```
key-indexed-count-sort(A, d)
A: array of size $n$, contains numbers with digits in $\{0, \ldots, R-1\}$
d: index of digit by which we wish to sort
// count how many of each kind there are
1.    count ← array of size $R$, filled with zeros
2.    for $i \leftarrow 0$ to $n-1$ do
3.        increment count[$d$th digit of $A[i]$]
// find left boundary for each kind
4.    idx ← array of size $R$, $idx[0] = 0$
5.    for $i \leftarrow 1$ to $R-1$ do
6.        $idx[i] \leftarrow idx[i-1] + count[i-1]$
// move to new array in sorted order, then copy back
7.    aux ← array of size $n$
8.    for $i \leftarrow 0$ to $n-1$ do
9.        $aux[idx[A[i]]] \leftarrow A[i]$
10.       increment $idx[A[i]]$
11.   $A \leftarrow copy(aux)$
```

Example: Count Sort

A		*count*	*idx*		*A*
$12\textcircled{3}$		$0, 1, 2, 3$	0		230
$23\textcircled{0}$		$0, 1, 2$	3		320
$02\textcircled{1}$		$0, 1$	5		210
$32\textcircled{0}$	$\Longrightarrow$	$0, 1$	6	$\Longrightarrow$	021
$21\textcircled{0}$					101
$23\textcircled{2}$					232
$10\textcircled{1}$					123

MSD-Radix-Sort

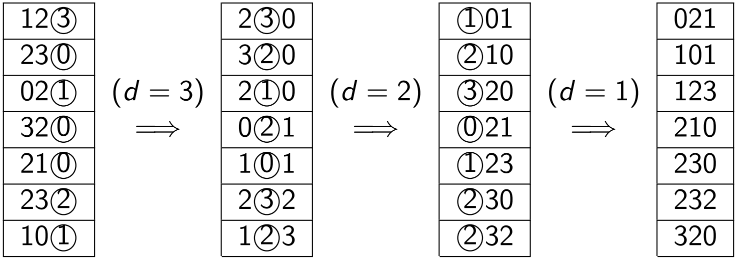
- Sorts multi-digit numbers.
- Obvious approach: sort by leading digit, then each group by next digit, etc.

```
$MSD\text{-}Radix\text{-}sort(A, l, r, d)$
$A$: array of size $n$, contains $m$-digit radix-$R$ numbers
$l, r, d$: integers, $0 \le l, r \le n-1$, $1 \le d \le m$
1.  if $l < r$
2.      partition $A[l..r]$ into bins according to $d$th digit
3.      if $d < m$
4.          for $i \leftarrow 0$ to $R-1$ do
5.              let $l_i$ and $r_i$ be boundaries of $i$th bin
6.              MSD-Radix-sort$(A, l_i, r_i, d+1)$
```

- Partition using count-sort.
- Drawback of MSD-Radix-Sort: many recursions

LSD-Radix-Sort

```
LSD-radix-sort(A)
A: array of size n, contains m-digit radix-R numbers
1.    for d ← m down to 1 do
2.        key-indexed-count-sort(A, d)
```



- Loop-invariant: A is sorted w.r.t. digits $d, \ldots, m$ of each entry.
- **Time cost**: $\Theta(m(n + R))$
- **Auxiliary space**: $\Theta(n + R)$

Summary

- Sorting is an important and *very* well-studied problem
- Can be done in $\Theta(n \log n)$ time; faster is not possible for general input
- HeapSort is the only $O(n \log n)$-time algorithm we have seen with $O(1)$ auxiliary space.
- MergeSort is also $\Theta(n \log n)$, selection & insertion sorts are $\Theta(n^2)$.
- QuickSort is worst-case $\Theta(n^2)$, but often the fastest in practice
- CountSort, RadixSort can achieve $o(n \log n)$ if the input is special

- Randomized algorithms can eliminate "bad cases"
- Best-case, worst-case, average-case, expected-case can all differ