VE281

Data Structures and Algorithms

Quick Sort; Comparison Sort Summary;

Non-comparison Sort

Outline

- Quick Sort
- Comparison Sort Summary
- Non-comparison Sort
  - Counting Sort

Review

- Quick sort: In-place partitioning
- Quick sort time complexity
  - Worst case: $O(N^2)$
  - Best case: $O(N \log N)$

Quick Sort

Average Case Time Complexity

- Average case time complexity of quick sort can be proved to be $O(N \log N)$.
  - Assume **randomly** pick an element from the array as pivot.
  - **Note**: average is over random choice of pivots made by the algorithm, **not** on the input.
  - The claim holds for any input.

Proof of Average Case Time Complexity

- Fix input array A of length N
- Sample space Ω: all possible pivot sequences that quick sort may choose
- Given random choice $\sigma \in \Omega$, define $C(\sigma)$= total number of comparisons made by quicksort
  - $C(\sigma)$ is a random variable
- Lemma: running time of quicksort is dominated by # of comparisons
  - I.e., there exists a constant c so that for all $\sigma \in \Omega$,
$$RunTime(\sigma) \leq c \cdot C(\sigma)$$
- Remaining goal: $E[C] = O(N \log N)$

Proof of Average Case Time Complexity

- Define $z_i = i$-th smallest element of A

3	6	5	2
z_2	z_4	z_3	z_1

- For each $\sigma \in \Omega$ and indices $i < j$,
  $X_{ij}(\sigma)$= # of times z_i, z_j get compared in quick sort with pivot sequence σ
- **Question**: what is the possible value of $X_{ij}(\sigma)$?
  - 0 or 1
  - **Reason**: two elements are compared only when one is the pivot. After that, they will not be compared any more

Proof of Average Case Time Complexity

- Important relation:

$$C(\sigma) = \sum_{i=1}^{N-1} \sum_{j=i+1}^{N} X_{ij}(\sigma)$$

- By linearity of expectation:

$$E[C(\sigma)] = \sum_{i=1}^{N-1} \sum_{j=i+1}^{N} E[X_{ij}(\sigma)]$$

0-1 random variable

- $E\left[X_{ij}(\sigma)\right] = \Pr(X_{ij} = 1)$
- Thus, $E[C(\sigma)] = \sum_{i=1}^{N-1} \sum_{j=i+1}^{N} \Pr(z_i, z_j \; get \; compared)$

Proof of Average Case Time Complexity

- Key claim: for all $i < j$,

$$\Pr(z_i, z_j \ get \ compared) = \frac{2}{j - i + 1}$$

- Proof of the key claim:
  - Fix z_i, z_j, consider the sequence $z_i, z_{i+1}, \dots, z_{j-1}, z_j$
  - As long as none of these are chosen as a pivot, all are passed to the same recursive call
  - Consider the first among $z_i, \dots, z_j$ that gets chosen as a pivot.

1. If z_i or z_j gets chosen first, then z_i and z_j are compared
2. If one of $z_{i+1}, \dots, z_{j-1}$ gets chosen first, then z_i and z_j are never compared: they are put into different recursive calls

Proof of Average Case Time Complexity

- Key claim: for all $i < j$,

$$\Pr\left(z_i, z_j\ get\ compared\right) = \frac{2}{j - i + 1}$$

- Proof of the key claim:
  1. If z_i or z_j gets chosen first, then z_i and z_j are compared
  2. If one of $z_{i+1}, \dots, z_{j-1}$ gets chosen first, then z_i and z_j are never compared
  - Since pivot sequence is chosen uniformly at random, each of $z_i, z_{i+1}, \dots, z_{j-1}, z_j$ is equally likely to be the first
  - Thus, $\Pr\left(z_i, z_j\ get\ compared\right) = \frac{2}{j-i+1}$

2: # choices lead to case 1

j-i+1: total # of choices

Proof of Average Case Time Complexity

- What we have so far: $E[C] = \sum_{i=1}^{N-1} \sum_{j=i+1}^{N} \frac{2}{j-i+1}$
- Our target: $E[C] = O(N \log N)$

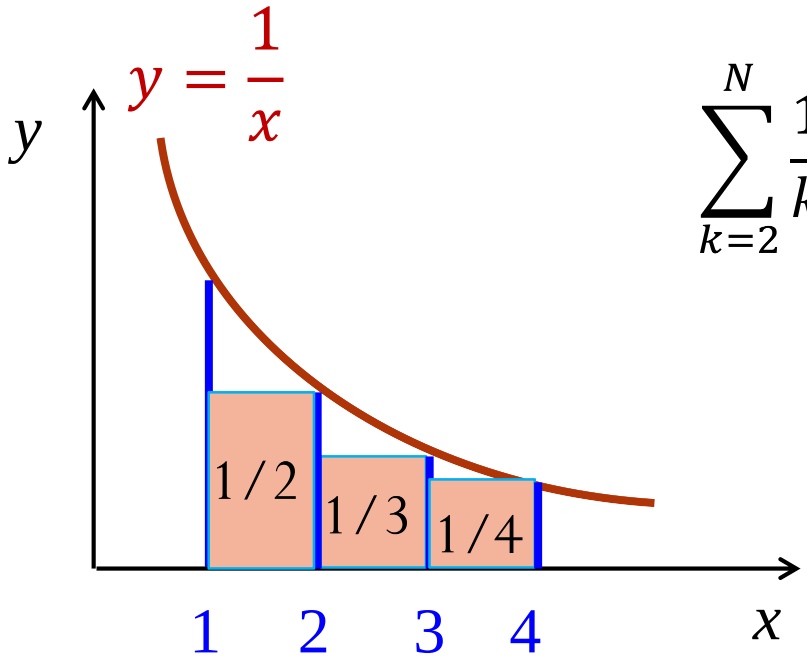
- Note: for each fixed $i \geq 1$,

$$\sum_{j=i+1}^{N} \frac{1}{j-i+1} \leq \sum_{j=i+1}^{N+i-1} \frac{1}{j-i+1} = \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{N}$$

- Claim: $\sum_{k=2}^{N} \frac{1}{k} < \ln N$
- Once we prove the above claim, we get $E[C] < 2N \ln N$

Proof of the Claim

- Claim: $\displaystyle\sum_{k=2}^{N}\frac{1}{k} < \ln N$



$$\sum_{k=2}^{N}\frac{1}{k} < \int_{1}^{N}\frac{1}{x}\,dx = \ln N$$

Quick Sort

Average Case Time Complexity

- Average case time complexity of quick sort is $O(N \log N)$.
  - Assume **randomly** pick an element from the array as pivot.
  - **Note**: average is over random choice of pivots made by the algorithm, **not** on the input.
  - The claim holds for any input.

Quick Sort

Other Characteristics

- In-place?
  - In-place partitioning.
  - Worst case needs $O(N)$ stack space.
  - Average case needs $O(\log N)$ stack space.
    - "Weekly" in-place.

- Not stable.

Quick Sort

Summary

- Like merge sort, quick sort is a divide-and-conquer algorithm.

- Merge sort: easy division, complex combination.
- Quick sort: complex division (partition with pivot step), easy combination.

- Insertion sort is faster than quick sort for small arrays.
  - Terminate quick sort when array size is below a threshold. Do insertion sort on subarrays.

Outline

- Quick Sort
- Comparison Sort Summary
- Non-comparison Sort
  - Counting Sort

Comparison Sorts

Summary

	Worst Case Time	Average Case Time	In Place	Stable
Insertion	$O(N^2)$	$O(N^2)$	Yes	Yes
Selection	$O(N^2)$	$O(N^2)$	Yes	No
Bubble	$O(N^2)$	$O(N^2)$	Yes	Yes
Merge Sort	$O(N \log N)$	$O(N \log N)$	No	Yes
Quick Sort	$O(N^2)$	$O(N \log N)$	Weakly	No

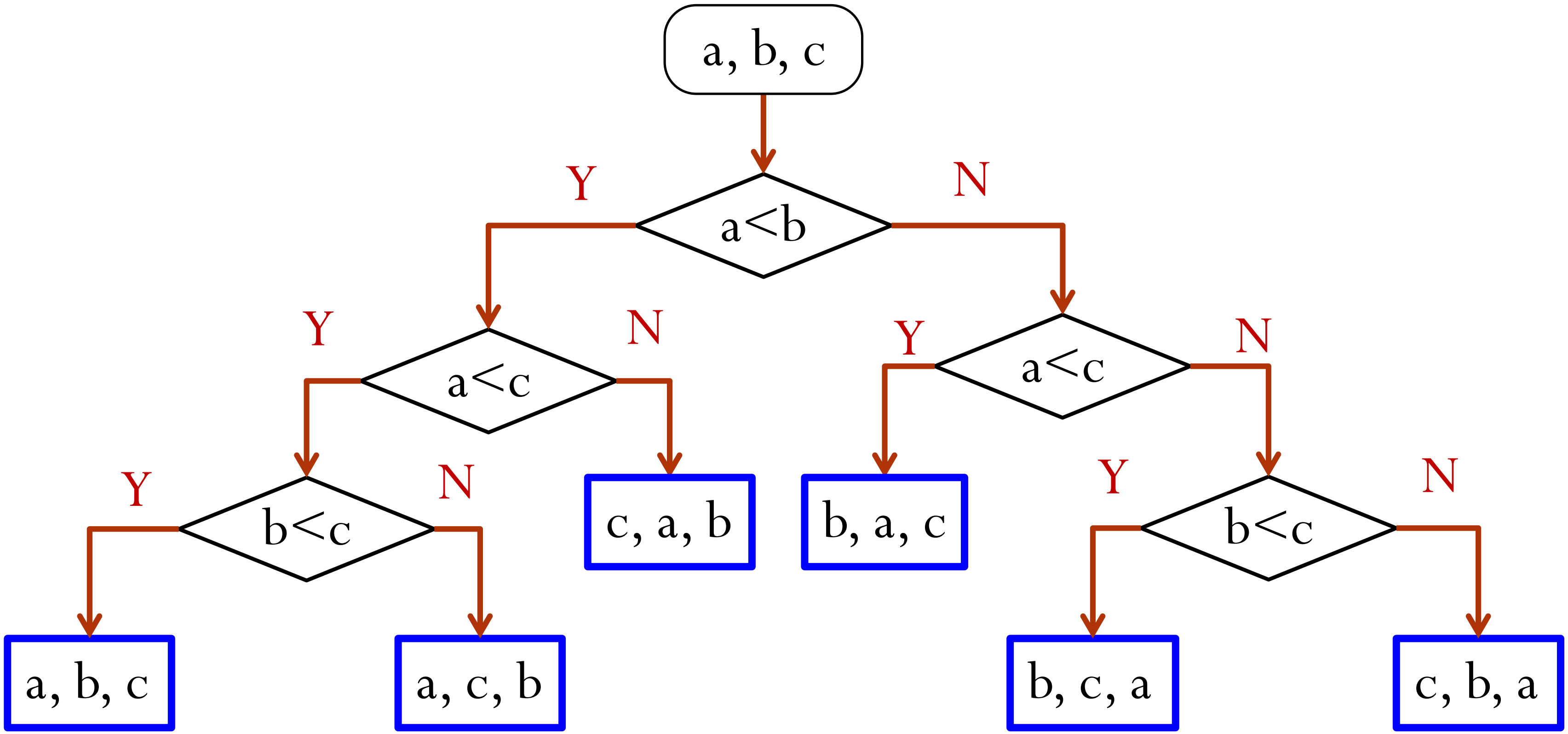
Comparison Sorts

Worst Case Time Complexity

- For comparison sort, is $O(N \log N)$ the best we can do in the worst case?
- Theorem: A sorting algorithm that is based on pairwise comparisons must use $\Omega(N \log N)$ operations to sort in the worst case.
- Proof: Consider the decision tree.

Decision Tree for 3 Items

- Input: an unsorted array of 3 items a, b, c.



Decision Tree and Theoretic Lower Bound

- Each sorting algorithm has a corresponding decision tree.
  - Decision tree is a binary tree.
- The sorting result is at one of the leaves following the results of a sequence of pairwise comparisons.
- The number of pairwise comparisons in the worst case corresponds to the deepest leaf in the decision tree, or the height of the tree.
- The number of leaves in a decision tree for sorting N items is $N!$, i.e., the number of permutations on N items.
- Note: a binary tree of height h has **at most** 2^h leaves. The height of the decision tree is **at least** $\lceil \log_2 N! \rceil$.

Theoretic Lower Bound

$$
\begin{aligned}
\log(N!) &= \log N + \log(N-1) + \cdots + \log 1 \\
&\geq \log N + \log(N-1) + \cdots + \log(N/2) \\
&\geq \frac{N}{2}\log(N/2) \\
&= \Omega(N \log N)
\end{aligned}
$$

- Thus, the worst case time complexity for comparison sorts is $\Omega(N \log N)$.
- Any way to beat the theoretic lower bound?
  - Do not compare keys: Non-comparison sort.

Outline

- Quick Sort
- Comparison Sort Summary
- Non-comparison Sort
  - Counting Sort

Counting Sort

A Simple Version

- Sort an array A of **integers** in the range $[0, k]$, where k is known.

1. Allocate an array **`count[k+1]`**.
2. Scan array A. For i**`=1`** to **`N`**, increment **`count[A[i]]`**.
3. Scan array **`count`**. For i**`=0`** to **`k`**, print **`i`** for **`count[i]`** times.

- Time complexity: $O(N + k)$.

- The algorithm can be converted to sort integers in some other known range $[a, b]$.
  - Minus each number by a, converting the range to $[0, b - a]$.

Counting Sort

A General Version

- In the previous version, we print **`i`** for **`count[i]`** times.
  - Simple but only works when sorting integer keys alone.
  - How to sort items when there is "additional" information with each key?
- A general version:

1. Allocate an array **`C[k+1]`**.
2. Scan array A. For i**`=1`** to **`N`**, increment **`C[A[i]]`**.
3. For **`i=1`** to **`k`**, **`C[i]=C[i-1]+C[i]`**
   - **`C[i]`** now contains number of items less than or equal to **`i`**.
4. For **`i=N`** downto **`1`**, put **`A[i]`** in new position **`C[A[i]]`** and decrement **`C[A[i]]`**.

Counting Sort

Example

1. Allocate an array **`C[k+1]`**.

k=5

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

2. Scan array A. For **`i=1`** to **`N`**, increment **`C[A[i]]`**.

	0	1	2	3	4	5
C	2	0	2	3	0	1

3. For **`i=1`** to **`k`**, **`C[i]=C[i-1]+C[i]`**

	0	1	2	3	4	5
C	2	2	4	7	7	8

4. For **`i=N`** downto **`1`**, put **`A[i]`** in new position **`C[A[i]]`** and decrement **`C[A[i]]`**.

Counting Sort

Example

1. Allocate an array **C[k+1]**.
2. Scan array A. For **i=1** to **N**, increment **C[A[i]]**.
3. For **i=1** to **k**, **C[i]=C[i-1]+C[i]**
4. For **i=N** downto **1**, put **A[i]** in new position **C[A[i]]** and decrement **C[A[i]]**.

k=5

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

	0	1	2	3	4	5
C	2	2	4	7	7	8

1	2	3	4	5	6	7	8
						3	

	0	1	2	3	4	5
C	2	2	4	6	7	8

Counting Sort

Example

1. Allocate an array `C[k+1]`.
2. Scan array A. For `i=1` to `N`, increment `C[A[i]]`.
3. For `i=1` to `k`, `C[i]=C[i-1]+C[i]`
4. For **`i=N`** downto **`1`**, put **`A[i]`** in new position **`C[A[i]]`** and decrement **`C[A[i]]`**.

k=5

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

	0	1	2	3	4	5
C	2	2	4	6	7	8

1	2	3	4	5	6	7	8
	0					3	

	0	1	2	3	4	5
C	1	2	4	6	7	8

Counting Sort

Example

1. Allocate an array `C[k+1]`.
2. Scan array A. For `i=1` to `N`, increment `C[A[i]]`.
3. For `i=1` to `k`, `C[i]=C[i-1]+C[i]`
4. For `i=N` downto `1`, put `A[i]` in new position `C[A[i]]` and decrement `C[A[i]]`.

k=5

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

	0	1	2	3	4	5
C	1	2	4	6	7	8

1	2	3	4	5	6	7	8
	0				3	3	

	0	1	2	3	4	5
C	1	2	4	5	7	8

Counting Sort

Example

1. Allocate an array `C[k+1]`.
2. Scan array A. For `i=1` to `N`, increment `C[A[i]]`.
3. For `i=1` to `k`, `C[i]=C[i-1]+C[i]`
4. For **`i=N`** downto **`1`**, put **`A[i]`** in new position **`C[A[i]]`** and decrement **`C[A[i]]`**.

k=5

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

	0	1	2	3	4	5
C	1	2	4	5	7	8

1	2	3	4	5	6	7	8
	0		2		3	3	

	0	1	2	3	4	5
C	1	2	3	5	7	8

Counting Sort

Example

1. Allocate an array `C[k+1]`.

2. Scan array A. For `i=1` to `N`, increment `C[A[i]]`.

3. For `i=1` to `k`, `C[i]=C[i-1]+C[i]`

4. **For `i=N` downto `1`, put `A[i]` in new position `C[A[i]]` and decrement `C[A[i]]`.**

k=5

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

	0	1	2	3	4	5
C	1	2	3	5	7	8

1	2	3	4	5	6	7	8
0	0		2		3	3	

	0	1	2	3	4	5
C	0	2	3	5	7	8

Counting Sort

Example

1. Allocate an array `C[k+1]`.
2. Scan array A. For `i=1` to `N`, increment `C[A[i]]`.
3. For `i=1` to `k`, `C[i]=C[i-1]+C[i]`
4. For `i=N` downto `1`, put `A[i]` in new position `C[A[i]]` and decrement `C[A[i]]`.

k=5

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

	0	1	2	3	4	5
C	0	2	3	5	7	8

1	2	3	4	5	6	7	8
0	0		2	3	3	3	

	0	1	2	3	4	5
C	0	2	3	4	7	8

Counting Sort

Example

1. Allocate an array `C[k+1]`.
2. Scan array A. For `i=1` to `N`, increment `C[A[i]]`.
3. For `i=1` to `k`, `C[i]=C[i-1]+C[i]`
4. For `i=N` downto `1`, put `A[i]` in new position `C[A[i]]` and decrement `C[A[i]]`.

k=5

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

	0	1	2	3	4	5
C	0	2	3	4	7	8

1	2	3	4	5	6	7	8
0	0		2	3	3	3	5

	0	1	2	3	4	5
C	0	2	3	4	7	7

Counting Sort

Example

1. Allocate an array `C[k+1]`.
2. Scan array A. For `i=1` to `N`, increment `C[A[i]]`.
3. For `i=1` to `k`, `C[i]=C[i-1]+C[i]`
4. For `i=N` downto `1`, put `A[i]` in new position `C[A[i]]` and decrement `C[A[i]]`.

k=5

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

	0	1	2	3	4	5
C	0	2	3	4	7	7

1	2	3	4	5	6	7	8
0	0	2	2	3	3	3	5

	0	1	2	3	4	5
C	0	2	2	4	7	7

Done!

Is counting sort stable?

Yes!