COT 5407: Introduction to Algorithms

Giri NARASIMHAN

www.cs.fiu.edu/~giri/teach/5407S19.html

Beating the Lower Bound

- Bucket Sort
  - Runs in time O(N+K) given N integers in range [a+1, a+K]
  - If K = O(N), we are able to sort in O(N)
  - How is it possible to beat the lower bound?
  - Only because we know more about the data.
  - If nothing is know about the data, the lower bound holds.
- Radix Sort
  - Runs in time O(d(N+K)) given N items with d digits each in range [1,K]
- Counting Sort
  - Runs in time O(N+K) given N items in range [a+1, a+K]

Bucket Sort

- N **integer** values in the range [a..a+m-1]
- For e.g., sort a list of 50 scores in the range [0..9].
- **Algorithm**
  - Make m buckets [a..a+m-1]
  - As you read elements throw into appropriate bucket
  - Output contents of buckets [0..m] in that order
- **Time O(N+m)**
- **Warning: <u>This algorithm cannot be used for "infinite-precision" real numbers, even if the range of values is specified.</u>**

Stable Sort

- A sort is stable if equal elements appear in the same order in both the input and the output.
- Which sorts are stable?

Radix Sort

3 5 9	3 5 9	3 3 6	3 3 6
3 5 7	3 5 7	3 5 9	3 5 1
3 5 1	3 5 1	3 5 7	3 5 5
7 3 9	3 3 6	3 5 1	3 5 7
3 3 6	3 5 5	3 5 5	3 5 9
7 2 0	7 3 9	7 2 0	7 2 0
3 5 5	7 2 0	7 3 9	8 3 9

Algorithm

for i = 1 **to** d **do**

    sort array A on digit i using any sorting algorithm

Time Complexity: $O((N+m) + (N+m^2) + \ldots + (N+m^d))$

Space Complexity: $O(m^d)$

Radix Sort

3 2 9		7 2 0		7 2 0		3 2 9
4 5 7		3 5 5		3 2 9		3 5 5
6 5 7		4 3 6		4 3 6		4 3 6
8 3 9	⇒	4 5 7	⇒	8 3 9	⇒	4 5 7
4 3 6		6 5 7		3 5 5		6 5 7
7 2 0		3 2 9		4 5 7		7 2 0
3 5 5		8 3 9		6 5 7		8 3 9

Algorithm

Time Complexity: O((n+m)d)

for i = 1 **to** d **do**

    sort array A on digit i using a stable sort algorithm

• **Warning**: This algorithm cannot be used for "infinite-precision" real numbers, even if the range of values is specified.

Counting Sort

Initial Array

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

Counts

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

Cumulative Counts

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

- **Warning**: This algorithm cannot be used for "infinite-precision" real numbers, even if the range of values is specified.

Tree Sorting

- BST is a search structure that helps efficient search
  - Search can be done in O(h) time, where h = height of BST
  - Also inserts and deletes can be done in O(h) time
  - Unfortunately, Height h = O(N)
- Balanced BST improves BST with h = O(log N)
  - Thus search can be done in O(log N)
  - And, inserts and deletes too can be done in O(log N) time
- We can use BBSTs in the following way:
  - Repeatedly insert N items into a BBST
  - Repeatedly delete the smallest item from the BBST until it is empty
- N inserts and N deletes can be done in O(N log N) time

Order Statistics

- Maximum, Minimum
  - Upper Bound
    - O(n) because ??
    - We have an algorithm with a single for-loop: n-1 comparisons
  - Lower Bound
    - n-1 comparisons
- MinMax
  - Upper Bound: 2(n-1) comparisons
  - Lower Bound: 3n/2 comparisons
- Max and 2ndMax
  - Upper Bound: (n-1) + (n-2) comparisons
  - Lower Bound: Harder to prove

7	3	1	9	4	8	2	5	

$\underline{\text{Rank}}_A(x)$ = position of x in sorted order o

k-Selection; Median

- Select the k-th smallest item in list
- Naïve Solution
  - Sort;
  - pick the k-th smallest item in sorted list.

    O(n log n) time complexity
- Idea: Modify Partition from QuickSort
  - How?
- Randomized solution: Average case O(n)
- Improved Solution: worst case O(n)

Using Partition for k-Selection

```
PARTITION(array A, int p, int r)
x ← A[r]                          ▷ Choose pivot
i ← p − 1
for j ← p to r − 1
        do if (A[j] ≤ x)
                then i ← i + 1
                        exchange A[i] ↔ A[j]
exchange A[i + 1] ↔ A[r]
return i + 1
```

- Perform Partition from QuickSort (assume all unique items)
- Rank(pivot) = 1 + # of items that are smaller than pivot
- If Rank(pivot) = k, we are done
- Else, recursively perform k-Selection in one of the two partitions

QuickSelect: a variant of QuickSort

```
QUICKSELECT(array A, int k, int p, int r)
  ▷ Select k-th largest in subarray A[p..r]
  if (p = r)
      then return A[p]
  q ← PARTITION(A, p, r)
  i ← q − p + 1        ▷ Compute rank of pivot
  if (i = k)
      then return A[q]
  if (i > k)
      then return QUICKSELECT(A, k, p, q)
      else return QUICKSELECT(A, k − i, q + 1, r)
```

k-Selection Time Complexity

- Perform Partition from QuickSort (assume all unique items)
- Rank(pivot) = 1 + # of items that are smaller than pivot
- If Rank(pivot) = k, we are done
- Else, recursively perform k-Selection in one of the two partitions

- On the average:
  - Rank(pivot) = n / 2
- Average-case time
  - T(N) = T(N/2) + O(N)
  - T(N) = O(N)
- Worst-case time
  - T(N) = T(N-1) + O(N)
  - T(N) = O(N^2)

```
PARTITION(array A, int p, int r)
x ← A[r]                          ▷ Choose pivot
i ← p − 1
for j ← p to r − 1
        do if (A[j] ≤ x)
              then i ← i + 1
                    exchange A[i] ↔ A[j]
exchange A[i + 1] ↔ A[r]
return i + 1
```

Randomized Solution for k-Selection

- Uses RandomizedPartition instead of Partition
  - RandomizedPartition picks the pivot uniformly at random from among the elements in the list to be partitioned.
- Randomized k-Selection runs in O(N) time on the average
- Worst-case behavior is very poor $O(N^2)$