VE281

Data Structures and Algorithms

Review Class

Oct. 29 2018

VE281 TA Group

Asymptotic Algorithm Analysis

- Best, Worst, Average Case
  - Best case: least number of steps required, corresponding to the ideal input
  - Worst case: most number of steps required, corresponding to the most difficult input.
  - Average case: average number of steps required, given any input.

Specific input. Not specific size!

Consider Large Input Size

Asymptotic Algorithm Analysis

- Big-Oh, Big Omega, Theta
  - Big-Oh:
    - $T(n) = O(f(n))$ if only if it exists two positive constants c and n_0 such that $T(n) \leq cf(n)$ for all $n > n_0$.
    - If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c < \infty$, then $f(n)$ is $O(g(n))$.
  - Big Omega: $T(n) \geq cg(n)$
  - Theta: $c_1 g(n) \leq T(n) \leq c_2 g(n)$

$$n! \gg 2^n \gg n^{c+\varepsilon} \gg n^3 \gg n^{2+\varepsilon} \gg n^2 \gg n \log n \gg n^{1+\varepsilon} \gg n$$

$$\gg \sqrt{n} \gg \log n \gg \frac{\log n}{\log \log n} \gg \log \log n \gg \alpha(n) \gg 1$$

Asymptotic Algorithm Analysis

- Analyzing Time Complexity of Programs

```
sum = 0;
for(i = 1; i <= n; i *= 2)
    for(j = 1; j <= i; j++)
        sum++;
```

- The number of times that the statements `sum++ / j<=i / j++` occur is

$$1 + 2 + 4 + 8 + \cdots 2^{\log n} \approx 2n - 1$$

- The time complexity is $\Theta(n)$.

Sorting

- Sorting Basics
  - In place: requires $O(1)$ additional memory
  - Stability: whether the algorithm maintains the relative order of records with equal keys
  - Comparison sort:

	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

Sorting

- Master Method: $\boldsymbol{T(n) \le aT\left(\frac{n}{b}\right) + O(n^d)}$

$$\boldsymbol{T(n) = \begin{cases} O\left(n^d \log n\right) & if\ a = b^d \\ O\left(n^d\right) & if\ a < b^d \\ O\left(n^{\log_b a}\right) & if\ a > b^d \end{cases}}$$

- Merge Sort
- Quick Sort
  - divide-and-conquer
  - $a = 2, b = 2, d = 1$

Sorting

- Counting Sort $O(N + k)$

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` down to `1`, put `A[i]` in new position `C[A[i]]` and decrement `C[A[i]]`.

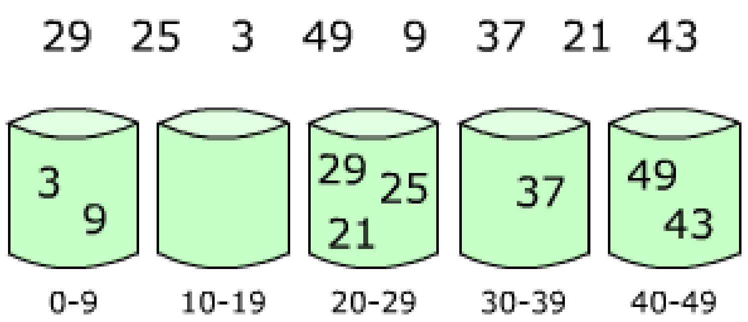
   `A[i]` is put in correct position

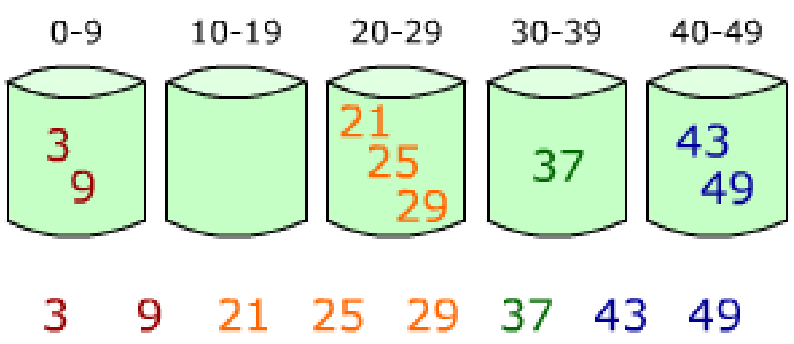
   since `C[A[i]]` items less than or equal to `A[i]`

Sorting

- Bucket Sort $O(N)$

1. Set up an array of initially empty "buckets".
2. Scatter: Go over the original array, putting each object in its bucket.
3. Sort each non-empty bucket by a comparison sort.
4. Gather: Visit the buckets in order and put all elements back into the original array.





Sorting

- Radix Sort $O(kN)$.

Given an array of integers, from the least significant bit (LSB) to the most significant bit (MSB), repeatedly do **stable** bucket sort according to the current bit

Sort 815, 906, 127, 913, 098, 632, 278.

0	1	2	3	4	5	6	7	8	9
		632	913		815	906	127	098 278	

0	1	2	3	4	5	6	7	8	9
906	913 815	127	632				278		098

0	1	2	3	4	5	6	7	8	9
098	127	278				632		815	906 913

Linear Time Selection

- Selection Problem:
  - Input: array A with n distinct numbers and a number i
  - Output: i-th smallest element in the array
- Solution:
  - Reduction to sorting $O(n \log n)$
  - Randomized selection algorithm $O(n)$
  - Deterministic selection algorithm $O(n)$

Linear Time Selection

- Randomized selection algorithm
  - Worst case: $O(n^2)$
  - Best case: $O(1)$
  - Average case: $O(n)$

$$E[N] = \frac{1}{2} \cdot 1 + \frac{1}{2} \cdot (1 + E[N])$$

$$E[runtime] \le E\left[\sum_j X_j \cdot c \cdot \left(\frac{3}{4}\right)^j n\right]$$

$$= cn \sum_j \left(\frac{3}{4}\right)^j E[X_j] \le 2cn \sum_j \left(\frac{3}{4}\right)^j \le 2cn \frac{1}{1 - \frac{3}{4}}$$

$$= 8cn = O(n)$$

Linear Time Selection

- Deterministic selection algorithm

```
ChoosePivot(A, n)
```

Time complexity: $O(n)$

$$T(n) \leq cn + T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) \leq 10n$$

Recursively find the median

reduce 30% of the size in each recursion

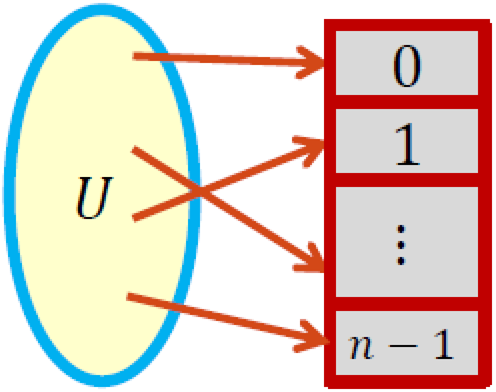
Not as good as Rselect in practice

Hash Basics

- Dictionary
  - **(key, value)**
  - <u>Different pairs have different keys</u>
  - Sorted/Unsorted array
  - find()
  - insert()
  - remove()

Hash Basics

- Hashing Basics
  - Pick an array A (hash table) of n buckets.
    - $n = c|S|$: a small multiple of $|S|$.
  - Choose a hash function $h: U \rightarrow \{0,1, \ldots, n-1\}$
  - Store item k in A[h(k)]
  - Runtime: $O(1)$



Hash Basics

- Hashing Function
  - Must compute a bucket for every key in the universe.
  - Must compute the same bucket for the same key.
  - Should be easy and quick to compute.
  - Minimizes collision
- Hashing by Modulo
  - The bias in the keys does not result in a bias toward home buckets.
  - Ideally, choose the hash table size n as a **large prime number**.

Hash Collision

- Collision:
  - Separate chaining: Link list (easy to remove())
  - Open addressing (save space)
    - Linear probing:
      `hi(x) = (h(x) + i) % n` — $h_i(x) = (h(x) + i) \% n$
    - Quadratic probing:
      $h_i(x) = (h(x) + i^2) \% n$
    - $L = \frac{m}{n} = \frac{\#\text{objects in hash table}}{\#\text{buckets in hash table}} \leq 0.5$ (find an empty slot)
    - Double hashing:
      $h_i(x) = (h(x) + i*g(x)) \% n$

Hash Collision

- Performance of Open Addressing:
  - Linear Probing
    - Find(): in sequence until find the key or find an empty slot
    - Remove(): rehash the cluster/remark “deleted”

$$U(L) = \frac{1}{2}\left[1 + \left(\frac{1}{1-L}\right)^2\right]$$

$$S(L) = \frac{1}{2}\left[1 + \frac{1}{1-L}\right]$$

  - Quadratic Probing

$$U(L) = \frac{1}{1-L}$$

$$S(L) = \frac{1}{L}\ln\frac{1}{1-L}$$