Algorithm and Data Structures
Lecture notes: Quicksort, Cormen, Chap. 7

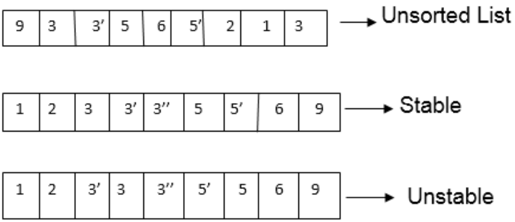
Lecturer: Michel Toulouse

Hanoi University of Science & Technology
michel.toulouse@soict.hust.edu.vn

25 mai 2020

Properties of sorting algorithms

- In place
  - Sorting of a data structure does not require any external data structure for storing the intermediate steps
  - Selection and insertion sort algorithms are ”In place”. Merge sort requires intermediary arrays
- Stable
  - If two objects with equal keys appear in the same order in sorted output as they appear in the input array to be sorted



Quicksort

Another divide-and-conquer algorithm

- The array A[p..r] is partitioned into two non-empty subarrays A[p..q] and A[q+1..r]
  - such that the elements in A[p..q] are less than all elements in A[q+1..r]
- The subarrays are recursively sorted by calls to quicksort
- Unlike merge sort, no combining step : two subarrays form an already-sorted array

Quicksort Code

```
Quicksort(A, p, r)
   if (p < r)
      q = Partition(A, p, r);
      Quicksort(A, p, q-1);
      Quicksort(A, q+1, r);
```

Sorts $O(n \log n)$ in the average case and $O(n^2)$ in the worst case

Partition

```
Quicksort(A, p, r)
  if (p < r)
    q = Partition(A, p, r) ;
    Quicksort(A, p, q-1) ;
    Quicksort(A, q+1, r) ;
```

- All the action takes place in the partition() function which rearranges the subarray
- End result : Two subarrays, all values in first subarray $\leq$ all values in second
- Returns the index of the "pivot" element separating the two subarrays

Partition

Partition(A, p, r) :

- Select an element to act as the "pivot" (which ?)
- Grow two regions, A[p..i] and A[j..r]
  - All elements in A[p..i] $\leq$ pivot
  - All elements in A[j..r] $>$ pivot
- Increment i until A[i] $>$ pivot
- Decrement j until A[j] $\leq$ pivot
- Swap A[i] and A[j]
- Repeat until $i \geq j$
- Return j

Partition code

```
Partition(A, p, r)
    x = A[p];
    i = p;
    j = r + 1;
    repeat i = i + 1 until A[i] > x or i ≥ j
    repeat j = j - 1 until A[j] ≤ x
    while i < j do
      Swap(A[i], A[j]);
      repeat i = i + 1 until A[i] > x
      repeat j = j -1 until A[j] ≤ x
    Swap(A[p], A[j]);
    return j;
```

Performance of Quicksort

- The running time of quicksort depends on the partitioning of the subarrays :
  - If the subarrays are balanced, then quicksort can run as fast as mergesort.
  - If they are unbalanced, then quicksort can run as slowly as insertion sort.

Worst case of Quicksort

Occurs when the subarrays are completely unbalanced. Have 0 element in one subarray and n .. 1 elements in the other subarray. Get the recurrence :

$$
\begin{aligned}
T(n) &= T(n-1) + T(1) + cn \\
&= [T(n-2) + T(1) + cn - 1] + T(1) + cn \\
&= T(n-2) + 2T(1) + c(n-1+n) \\
&= [T(n-3) + T(1) + cn - 2] + 2T(1) + c(n-1+n) \\
&= T(n-i) + iT(1) + c(n-i+1+\ldots+n-2+n-1+n) \\
&= T(n-i) + iT(1) + c(\sum_{j=0}^{i-1}(n-j)) \text{ substituting } i \text{ by } n-1 \\
&= T(1) + (n-1)T(1) + c\sum_{j=0}^{n-2}(n-j) \\
&= nT(1) + \theta(n^2) \\
&= \Theta(n^2)
\end{aligned}
$$

The same running time as insertion sort. Worst-case running time occurs when quicksort takes a sorted array as input, insertion sort runs in $O(n)$ time in this case.

Best case of Quicksort

- Occurs when the subarrays are completely balanced every time.
- Each subarray has $\leq n/2$ elements.
- Get the recurrence :

$$\begin{aligned} T(n) &= 2T(n/2) + \Theta(n) \\ &= \Theta(n \log n) \end{aligned}$$

Quicksort well designed

It is easy to avoid the worst case performance for quicksort.

Your textbooks introduces two possible solutions to this issue :

- Randomize the input, or
- Pick a random pivot element

This solve the bad worst-case behavior because no particular input can be chosen that makes quicksort runs in $O(n^2)$

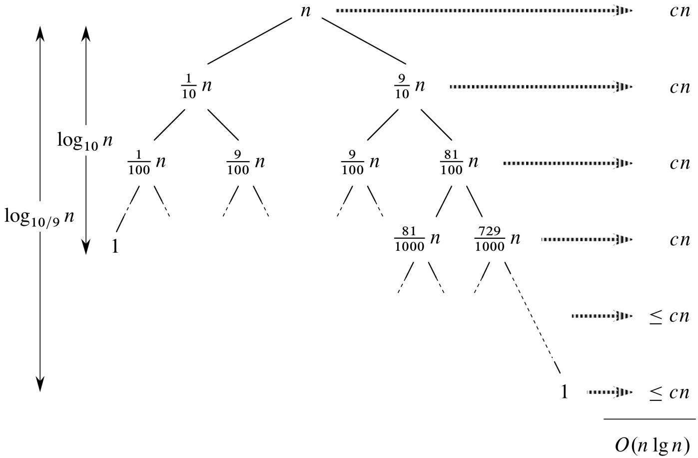
Analyzing quicksort : effects of partitioning

- Quicksort's average running time is much closer to the best case than to the worst case.
- Imagine that PARTITION always produces a 9-to-1 split.
- Get the recurrence :

$$\begin{aligned} T(n) &\leq T(9n/10) + T(n/10) + \Theta(n) \\ &= O(n \log n) \end{aligned}$$

- See the recurrence tree in the next slide, the number of levels of the partitioning is $\log_{\frac{10}{9}} n$, local work at each level cost no more than cn, therefore quicksort runs in $O(n \log n)$ even for this bad partitioning

Analysing quicksort : effects of partitioning



Analyzing Quicksort : Average Case

- Assumption : Select randomly the partition element
- all splits (0 :n-1, 1 :n-2, 2 :n-3, ..., n-1 :0) equally likely
- therefore, each split has probability 1/n to occur
- if T(n) is the expected running time

$$
\begin{aligned}
T(n) &= \frac{1}{n} \sum_{k=0}^{k=n-1} (T(k) + T(n-1-k) + O(n)) \\
&= \frac{2}{n} \sum_{k=0}^{k=n-1} (T(k) + O(n))
\end{aligned}
$$

Analyzing Quicksort : Average Case

- We can solve this recurrence using the substitution method
- Guess the answer T(n) = O(n lg n)
- Assume that the inductive hypothesis holds
  - $T(n) \leq cn \log n$ for some constant c
- Substitute it in for some value < n
  - The value k in the recurrence

  Prove that it follows for n

Analyzing Quicksort : Average Case

$$
\begin{aligned}
T(n) &= \frac{2}{n} \sum_{k=0}^{k=n-1} T(k) + O(n) \\
&\leq \frac{2}{n} \sum_{k=0}^{k=n-1} (ck \log k) + O(n) \text{ inductive hypothesis} \\
&= \frac{2c}{n} \sum_{k=0}^{k=n-1} k \log k + O(n) \\
&\leq \frac{2c}{n} \left(\frac{1}{2} n^2 \log n - \frac{1}{8} n^2 \right) + O(n)
\end{aligned}
$$

The summation $\sum_{k=1}^{k=n-1} k \log k$ can be bound above by $\frac{1}{2} n^2 \log n - \frac{1}{8} n^2$ but this is not proved here.

Analyzing Quicksort : Average Case

$$
\begin{aligned}
T(n) &\leq \frac{2c}{n}\left(\frac{1}{2}n^2\log n - \frac{1}{8}n^2\right) + O(n) \\
&= cn\log n - \frac{c}{4}n + O(n) \\
&= cn\log n + \left(O(n) - \frac{c}{4}n\right) \\
&= cn\log n \text{ for } c \text{ such that } \tfrac{cn}{4} > O(n)
\end{aligned}
$$

Thus $T(n) \in O(n\log n)$