Lecutre 4 Quicksort & Selection

By Gou Guanglei

1. Quicksort

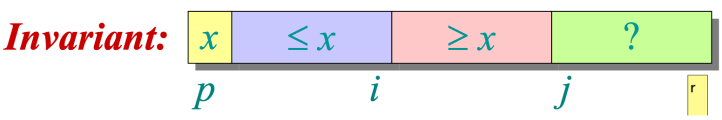
- Quicksort was discovered by Tony Hoare in 1962.
- The hard work is splitting the array into subsets so that merging the final result is trivial.
    - **Divide:** Partition the array into two subarrays around a ***pivot*** x such that elements in lower subarray$\leq x$, x , $x \leq$ elements in upper subarray. (Sorts **"in place"**)

    - **Conquer:** Recursively sort the two subarrays.
    - **Combine:** Trivial. (No work need to do becuase the subarrays are already sorted.)

```
QUICKSORT(A, p, r)
  if p < r
    q = PARTITIION(A, p, r)
    QUICKSORT(A, p, q - 1)
    QUICKSORT(A, q + 1, r)
```

```
1 PARTITION(A, p, r)
2   x = A[p]
3   i = p
4   for j = p + 1 to r
5     if A[j] <= x
6       i = i + 1
7       exchange A[i] with A[j]
8   exchange A[p] with A[i]
9   return i
```

- At the beginning of each iteration of the loop in `PARTITION` of lines 3-6, for any array index k,
    - **Condition 1**: If $p \leq k \leq i,$ then $A[k] \leq x$
    - **Condition 2**: If $i+1 \leq k \leq j-1,$ then $A[K] > x$
    - **Condition 3**: If $k = p$, then $A[k] = x$



1.1 Proof of correctness for Quicksort

- The proof of correctness for `QUICKSORT` is actually two proofs, a proof that `PARTITION` does the right thing and a proof that `QUICKSORT` does the right thing.
    - The proof of correctness for Partition is a fairly standard proof involving a loop invariant. The appropriate invariant is $A[p\,..\,i] \le x$ and $A[i+1..j-1] > x$
        - **Initialization**: Prior to the first iteration of the loop, $i = p$ and $j = p + 1$. There is no element lie between p and j and no element lie between $i + 1$ and $j - 1$. The first two conditions of the loop invariant are trivially satisfied. The assign- ment in line 1 satisfies the **condition 3**.
        - **Maintenance:** There are two cases.
            - The only action in the loop is to increment j when $A[j] > x$. After j is incremented, **condition 2** holds for $A[j-1]$ and other entries remain unchanged.
            - The action is to the loop increment i , swap $A[i]$ and $A[j]$, and then increment j when $A[j] \le x$. Because of the swap, we now have that $A[i] \le x$ and **condition 1** is satisfied.
            - **Termination:** At termination, $j = r$. Therefore, every entry in the array is in one of the three sets described by the invariant, and **we have partitioned the values in the array into three sets**: those less than or equal to x, those greater than x, and a singleton set containing x.
    - **Theorem**. `QUICKSORT(A,p,r)` can correctly sort $A[p\,..\,r]$ in ascending order.
        - When *p* = *r* Quicksort does nothing, which is the correct thing to do when sorting an array of length 1.
        - Assuming that Quicksort can correctly sort any array of length n or less, we show that it can correctly sort an array of length $n + 1$. The partition step that comes first will partition the array of length $n + 1$ into two subarrays and a **pivot**. That is, we end up with $A[p\,..\,q-1] \le A[q] < A[q+1..r]$. The largest of these subarrays will have length n or less, so the induction hypothesis tells us that the recursive calls to Quicksort will correctly sort the two subarrays. Once the subarrays are sorted we are done.

1.2 Performance of Quicksort

Best-case paritition

- `PARTITION` split the array evenly
- $T(n) = 2T(n/2) + \Theta(n) = \Theta(nlogn)$

Worst-case partition

- `PARTITION` split the array into one with $n - 1$ elements and one with 0 element.
- $T(n) = 2T(n-1) + T(0) + \Theta(n) = T(n-1) + \Theta(n) = \Theta(n^2)$

1.3 Quicksort in practice

- Quicksort is a great general-purpose sorting algorithm.
- Quicksort is typically over twice as fast as merge sort.

- Quicksort can benefit substantially from code tuning.
- Quicksort behaves well even with caching and virtual memory.

2. Randomized Quicksort

2.1 Randomized quick sort

- **IDEA:** Partition around a ***random*** element.
  - Running time is independent of the input order.
  - No assumptions need to be made about the input distribution.
  - No specific input elicits the worst-case behavior.
  - The worst case is determined only by the output of a random-number generator.

```
RANDOMIZED-PARTITION(A, q, r)
  i = RANDOM(p, r)
  exchange A[r] with A[i]
  return PARTITION(A, p, r)
```

```
RANDOMIZED-QUICKSORT(A, p, r)
  if p < r
    q = RANDOMIZED-PARTITION(A, q, r)
    RANDOMIZED-QUICKSORT(A, p, q - 1)
    RANDOMIZED-QUICKSORT(A, q + 1, r)
```

2.2 Analysis of Random quick sort

1. Worst-case analysis

- $T(n) = \max_{0 \leq q \leq n-1}(T(q) + T(n-q-1)) + \Theta(n)$
  - Guess $T(n) \leq cn^2$
  - $T(n) \leq \max_{0 \leq q \leq n-1}(cq^2 + c(n-q-1)^2) + \Theta(n) = c \cdot \max_{0 \leq q \leq n-1}(q^2 + (n-q-1)^2) + \Theta(n)$

    hence, $\max_{0 \leq q \leq n-1}(q^2 + (n-q-1)^2) \leq (n-1)^2 = n^2 - 2n + 1$.

    Therefor, $T(n) \leq cn^2 - c2(n-1) + \Theta(n) \leq cn^2$
  - $T(n) = \Theta(n^2)$

2. Expected running time analysis

- For $k = 0, 1, \dots, n-1$, define the indicator random variable
- $F_n = \begin{cases} 1 & \text{if PARTITION generates a k:n-k-1 split} \\ 0 & \text{otherwise} \end{cases}$
- $E[X_k] = Pr\{X_k = 1\} = 1/n$, since all splits are equally likely, assuming elements are distinct.

$$T(n) = \begin{cases} T(0) + T(n-1) + \Theta(n) & \text{if 0:n-1 split} \\ T(1) + T(n-2) + \Theta(n) & \text{if 1:n-2 split} \\ \dots \\ T(n-1) + T(0) + \Theta(n) & \text{if n-1:0 split} \end{cases}$$

$T(n) = \sum_{k=0}^{n-1} X_k(T(k) + T(n-k-1) + \Theta(n))$

- Take expectations of both sides

$$\begin{aligned} E[T(n)] &= E[\sum_{k=0}^{n-1} X_k(T(k) + T(n-k-1) + \Theta(n))] \\ &= \sum_{k=0}^{n-1} E[X_k(T(k) + T(n-k-1) + \Theta(n))] \\ &= \sum_{k=0}^{n-1} E[X_k] \cdot E[T(k) + T(n-k-1) + \Theta(n)] & (\textstyle\sum_{k=0}^{n-1} E[X_k] = 1/n) \\ &= (1/n) \sum_{k=0}^{n-1} E[T(k)] + (1/n) E[T(n-k-1)] + (1/n) \sum_{k=0}^{n-1} \Theta(n)] \\ &= (2/n) \sum_{k=1}^{n-1} E[T(k)] + \Theta(n) \end{aligned}$$

- $E[T(n)] = (2/n) \sum_{k=2}^{n-1} E[T(k)] + \Theta(n)$ (The $k = 0, 1$ terms can be absorbed in the $\Theta(n)$)
- $E[T(n)] \le anlgn$ for constant $a > 0$
  - use fact $\sum_{k=2}^{n-1} klgk \le (1/2)n^2 lgn - (1/8)n^2$
  - $$\begin{aligned} E[T(n)] &\le (2/n) \sum_{k=2}^{n-1} aklgk + \Theta(n) \\ &= (2a/n)((1/2)n^2 lgn - (1/8)n^2) + \Theta(n) \\ &= anlgn - an/4 + \Theta(n) \\ &\le anlgn \end{aligned}$$
- **Conlusion**: $T(n) = O(nlgn)$

3. Another expected running time analysis

- The running time of `QUICKSORT` is dominated by the time spent in the `PARTITION` procedure.
  - Each time the `PARTITION` procedure is called, it selects a pivot element,
  - and this element is **never included in any future recursive calls** to `QUICKSORT` and `PARTITION`.
- There can be **at most *n* calls** to `PARTITION` over the entire execution of the quicksort algorithm.
- One call to `PARTITION` takes $O(1)$ time plus an amount of time that is proportional to the number of iterations of the **for** loop in lines 3–6.
- Each iteration of this **for** loop performs a comparison in line 4, comparing the pivot element to another element of the array A.
- Therefore, if we can count the total number of times that line 4 is executed, we can bound the total time spent in the **for** loop during the entire execution of `QUICKSORT`.

Lemma

- Let X be the number of comparisons performed in line 4 of `PARTITION` over the entire execution of `QUICKSORT` on an n-element array. Then the running time of QUICKSORT is $O(n + X)$.

- $Proof$. By the discussion above, the algorithm makes at most n calls to `PARTITION`, each of which does a constant amount of work and then executes the **for** loop some number of times. Each iteration of the **for** loop executes line 4.
- We will derive an overall bound on the **total number of comparisons** rather than analyzing how many comparisons are made in *each* call to `PARTITION`.
- Elements are **compared only to the pivot** element and, after a particular call of `PARTITION` finishes, the pivot element used in that call is **never again compared** to any other elements.
  - Define the set $Z_{ij} = z_i, z_{i+1}, \dots, z_j$ to be the set of element between z_i and z_j, where $z_1, z_2, \dots, z_n$ are rename the elements of the array A.
  - Define $X_{ij} = I\{z_i \text{ is compared to } z_j\}$

    $X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$ (each pair is compared at most once)
  - Taking expectations of both sides,

    $$E[X] = E[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[X_{ij}] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} Pr\{z_i \text{ is compared to } z_j\}$$

---

example : Consider a `QUICKSORT` of the numbers 1 through 10 and suppose the pivot element is 7. After the call to `PARTITION`, the numbers are split into two sets $\{1, 2, 3, 4, 5, 6\}$ and $\{8, 9, 10\}$. (in any order)

- The pivot 7 is compared to all other elements
- No number from the set $\{1, 2, 3, 4, 5, 6\}$ will be compared to any number from the set $\{8, 9, 10\}$

- Assuming that element values are distinct, **once a pivot x is chosen** with $z_i < x < z_j$, z_i and z_j **cannot be compared at any subsequent time**.
  - $Z_{7,9}$ in the example, 7 and 9 are compared because 7 is pivot.
  - $Z_{2,9}$, 2 and 9 will never be compared because the pivot 7.
- Thus, z_i and z_j are compared **if and only if** a pivot is either z_i or z_j from Z_{ij} .
- Any element of Z_{ij} is equally likely to be the pivot.
- Z_{ij} has $j - i + 1$ elements.
- Therefore, the probability is $1/(j - i + 1)$ for choosing one element as a pivot in Z_{ij}.

  $$\begin{aligned} Pr\{z_i \text{ is compared to } z_j\} &= Pr\{z_i \ or \ z_j \text{ is the pivot chosen from } Z_{ij}\} \\ &= Pr\{z_i \text{ is the pivot chosen from } Z_{ij}\} + Pr\{z_j \text{ is the pivot chosen from } Z_{ij}\} \\ &= 1/(j - i + 1) + 1/(j - i + 1) \\ &= 2/(j - i + 1) \end{aligned}$$

  $$\begin{aligned} E[X] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} 2/(j - i + 1) \quad (\ k = j - i\ ) \\ &= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} 2/(k + 1) \\ &< \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} 2/k \end{aligned}$$

$= \sum_{i=1}^{n-1} O(lgn)$

$= O(nlgn)$

- Thus we conclude that, using `RANDOMIZED-QUICKSORT`, the expected running time of quicksort is $O(nlgn)$ when element values are distinct.

3. Randomized Selection

- returns the i-th smallest element of the array
- Divide and conquer algorithm for the selection proble.

```
RANDOMIZED-SELECT (A, p, r, i)
  if p == r
    return A[p]
  q = RANDOMIZED-PARTITION (A, p, r)
  k = q - p + 1
  if i == k
    return A[q]
  else if i < k
    return RANDOMIZED-PARTITION (A, p, q - 1, i)
  else
    return RANDOMIZED-PARTITION (A, q + 1, r, i - k)
```

3.1 Worst-case analysis

- $T(n) = \Theta(n^2)$
- If the pivot *p* is chosen to be the **minimum** or **maximum** element, then `RANDOMIZED-SELECT` runs in $\Theta(n^2)$.

3.2 Expected running time analysis

- The analysis follows that of randomized quicksort, but it's a little different.

$$X_k = \begin{cases} 1 & \text{if PARTITION generates a k:n-k-1 split} \\ 0 & \text{otherwise} \end{cases}$$

$$T(n) = \begin{cases} T(max\{0, n-1\}) + \Theta(n) & \text{if 0:n-1 split} \\ T(max\{1, n-2\}) + \Theta(n) & \text{if 1:n-2 split} \\ \ldots & \\ T(max\{n-1, 0\} + \Theta(n) & \text{if n-1:0 split} \end{cases}$$

$$T(n) = \sum_{k=0}^{n-1} X_k(T(max\{k, n-k-1\}) + \Theta(n))$$

$$E[T(n)] = E[\sum_{k=0}^{n-1} X_k(T(max\{k, T(n-k-1\}) + \Theta(n))]$$

$$= (1/n) \sum_{k=0}^{n-1} E[T(max\{k, n-k-1\})] + (1/n) \sum_{k=0}^{n-1} \Theta(n)]$$

$$\leq (2/n) \sum_{k=\lfloor n/2 \rfloor}^{n-1} E[T(k)] + \Theta(n)$$

$Proof.$ $E[T(n)] \leq cn$. Use fact $\sum_{k=\lfloor n/2 \rfloor}^{n-1} k \leq (3/8)n^2$

- If the pivot *p* is chosen to be the **median** element,, then `RANDOMIZED-SELECT` runs in $O(n)$.

3. Worst-case Linear-time Selection

3.1 Worst-case Linear-time Selection Algorithm

- IDEA: Generate a *good* pivot *recursively*.
- Find a pivot "close enough" to the median

```
CHOOSEPIVOT(A, n)
  Split A into m = ceil(n/5) groups p1, p2, ... ,pm
  medians[m]
  for i = 1 to m
    sort(pi)
    medians[i] = pi[2]                          //the median of sorted pi, 5/2 = 2
  mom = RANDOMIZED-SELECT (medians, 1, m, m/2)    // the median of medians, m/2
  return mom
```

3.2 Asymptotic analysis

- Half of the *median* $(\lceil m/2 \rceil - 1)$ is larger / less than **mom** in the *medians*.
- There are **at lest** $3 \cdot (\lceil m/2 \rceil - 2)$ elements are larger/less than mom.
- $m = \lceil n/5 \rceil$
- At least $(3n/10 - 6)$ are larger / less than mom.
- Thus, in the **worst case**, we RANDOMIZED−SELECT recursively on **at most** $n - 1 - (3n/10 - 6) = 7n/10 + 6$ elements.
- Running time $T(n)$ = Find mom $T(n/5)$ + At most select $T(7n/10)$
- $T(n) \le T(n/5) + T(7n/10) + O(n)$
  - Suppose $T(n) \le cn$. Using substitution method
  - $T(n) \le c\lceil n/5 \rceil + c(7n/10) + an \le cn/5 + 7cn/10 + an = cn + (-cn/10 + an)$