UNIVERSITAS INDONESIA
Veritas, Probitas, Iustitia

FAKULTAS ILMU KOMPUTER

Randomized Algorithm

Monte Carlo, Las Vegas, Quicksort, Randomized Quicksort,

(Arlisa Yuliawati)

Introduction

- Algorithm design patterns:
  - Incremental improvement (iterative algorithm)
  - Divide and Conquer
  - **Randomization**
    - It can lead to simplest, fastest algorithm, or only known algorithm for a particular problem.
    - Example: Symmetry breaking protocols, graph algorithms, quicksort, hashing, load balancing, Monte Carlo integration, cryptography, matrix multiplication verifier.
  - Greedy
  - Dynamic Programming
  - Network Flow

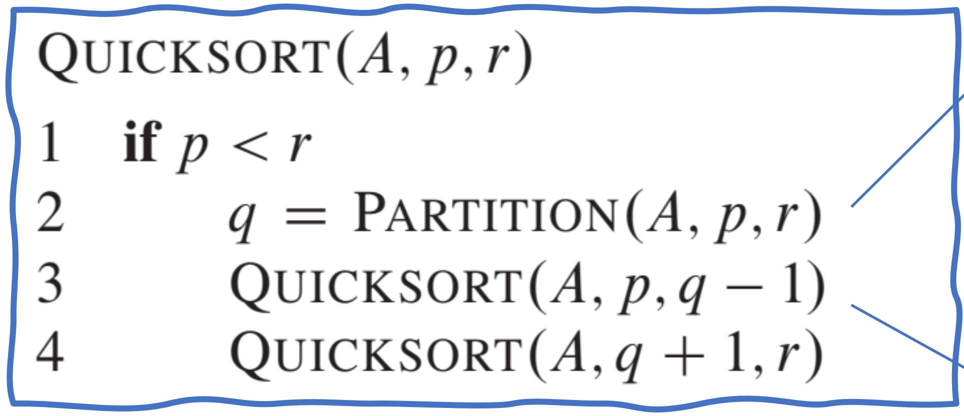
Randomized Algorithm

- Randomized algorithms are the algorithms whose behavior is not only determined by their input, but also by values produced by a random-number generator. (CLRS)
  - **Monte Carlo algorithms:** guaranteed to run in polynomial time, likely to find the correct answer. It has **fixed running time**, but its correctness is random.
    - Example: **Freivalds' matrix multiplication verifier**
  - **Las Vegas algorithms:** guaranteed to find the correct answer, likely to run in polynomial time. **It is always correct**, but the running time is a random variable.
    - Example: **Randomized quicksort**, Johnson's MAX-3SAT algorithm.

Quicksort and Its Analysis

Quicksort

- Quicksort is an in-place sorting algorithm
- It works based on divide and conquer approach



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

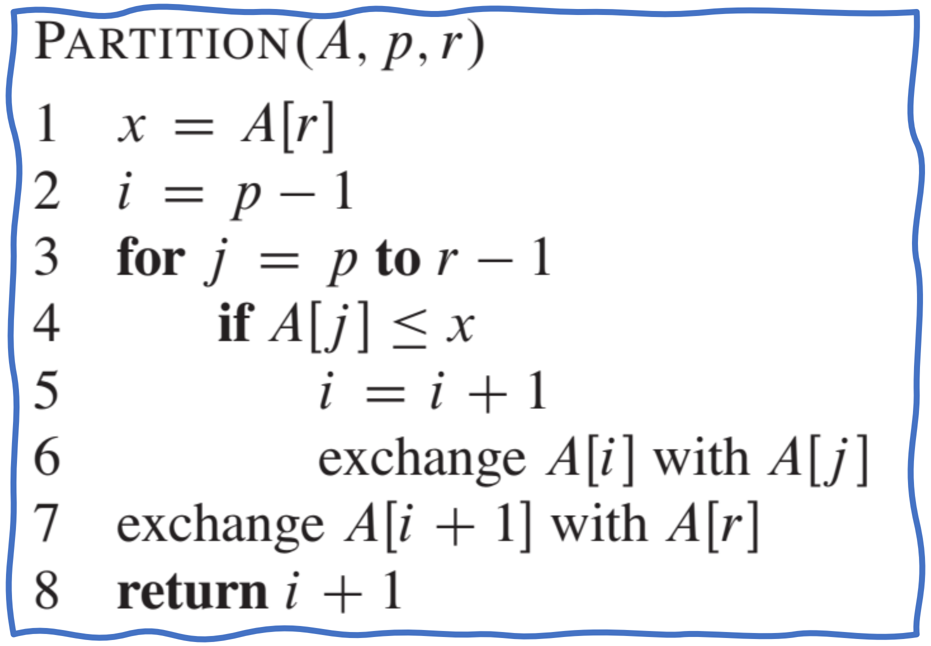
Divide: partition into 2 subarrays (A[p…q-1] ≤ A[q] and A[q+1…r] > A[q])

Conquer: recursively sort the two subarrays A[p…q-1] and A[q+1…r]

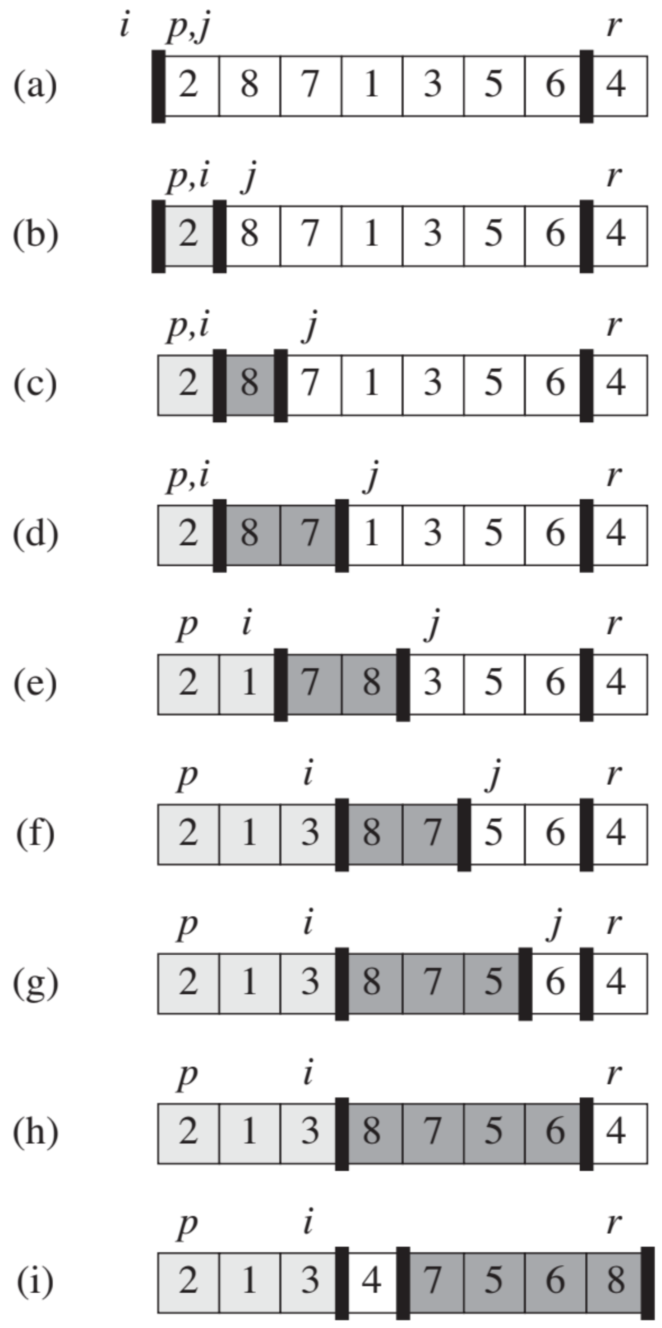
No work is needed to combine the subarrays

Quicksort

- Lomuto Partition



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



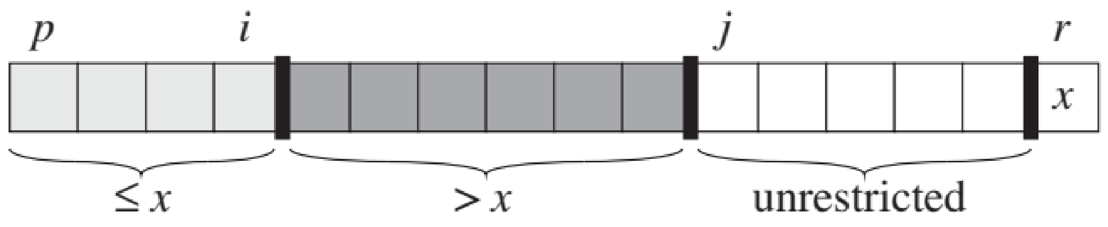
Correctness

- The main process of Quicksort is Partition procedure. The following is loop invariant to show its correctness.

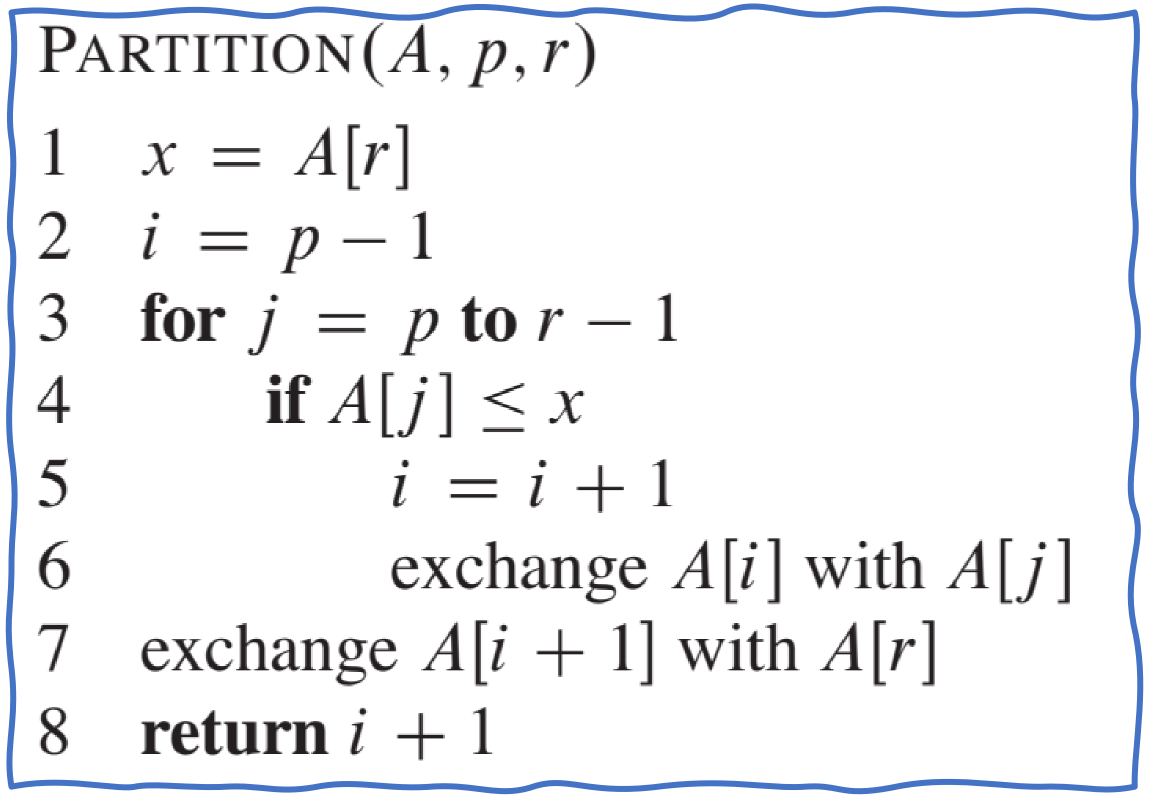
At the beginning of each iteration of the loop of lines 3–6, for any array index k,

1. If $p \le k \le i$, then $A[k] \le x$.
2. If $i + 1 \le k \le j - 1$, then $A[k] > x$.
3. If $k = r$, then $A[k] = x$.

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



Running Time of PARTITION



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

- It is dominated by the for-loop in line 3-6.
  - It runs from j = p to r-1.
  - The largest array being executed is the input array of length n (the first call to PARTITION procedure) where p = 1 and r = A.length = n
- Since the cost for line 1-2 and 7-8 are constant, the total running time is $\Theta(n)$.

Performance

- The running time of quick sort is determined by the partition result.
  - **Best case** partitioning

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$$

  - **Worst case** partitioning

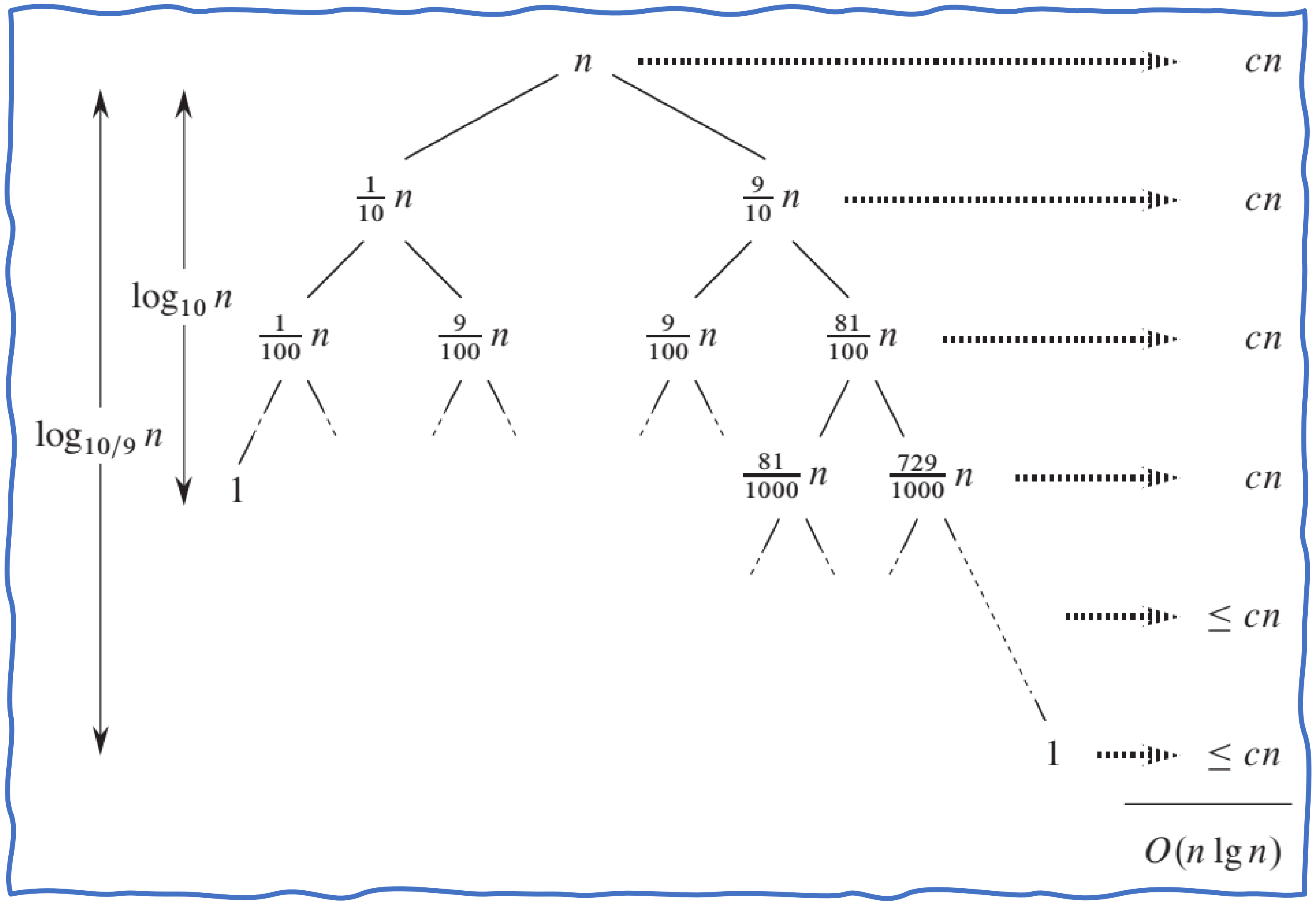
$$T(n) = T(n-1) + \Theta(n)$$

  - **Average case** partitioning
    - Proportional split
    - Randomized Quicksort

Proportional Split

- When the partition algorithm always produce c to $d-c$ proportional split. For example:

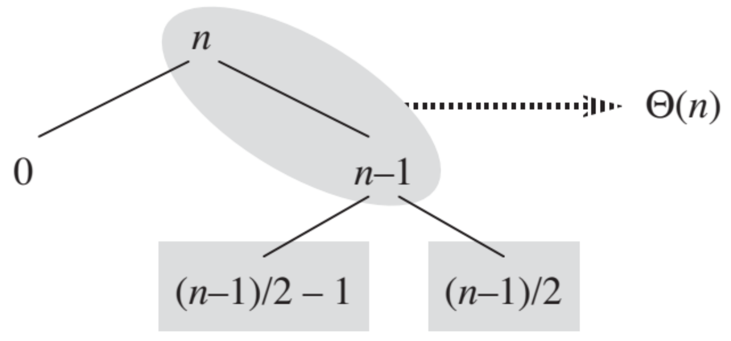
- $T(n) = T\left(\frac{9n}{10}\right) + T\left(\frac{n}{10}\right) + \Theta(n)$



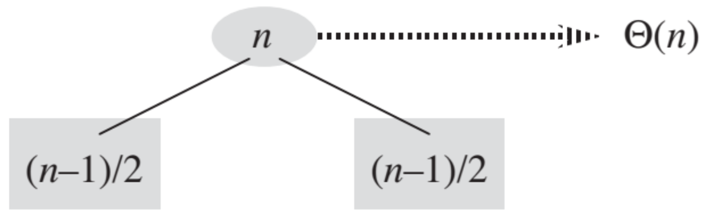
Intuition for Average Case

- Basically, the behavior of Quick Sort depends on the <u>relative ordering of the elements in an array</u>, not by the particular values in the array.
- The characteristic of the split on each level of the recursion tree is <u>highly unlikely to be the same</u>.
  - Some splits will be reasonably well balanced, and some others will be fairly unbalanced
- In the average case, PARTITION produces <u>a mix of good and bad splits</u>, and it is distributed randomly throughout the tree.

Intuition for Average Case



(a)



(b)

- Case (a) is no worse than case (b)
  - The split for n-1 elements in (a) is similar to the split in (b)
  - The total cost for partitioning in (a) is $\Theta(n) = \Theta(n) + \Theta(n-1)$
- The running time for (a) is like the running time for (b): $O(n \lg n)$
  - But with a slightly larger constant hidden by the O-notation.

Randomized Quicksort

- A **Las Vegas** version of Quicksort
- It randomly choose an element from the subarray A[p…r] as the pivot.
- We expect the splits to be reasonably well-balanced on average.
- Under the assumption that <u>all permutations of the input numbers are equally likely</u>, and the values of elements being sorted are distinct.

- Based on previous intuition for average running time, it going to need $O(n \lg n)$

Randomized Quicksort

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

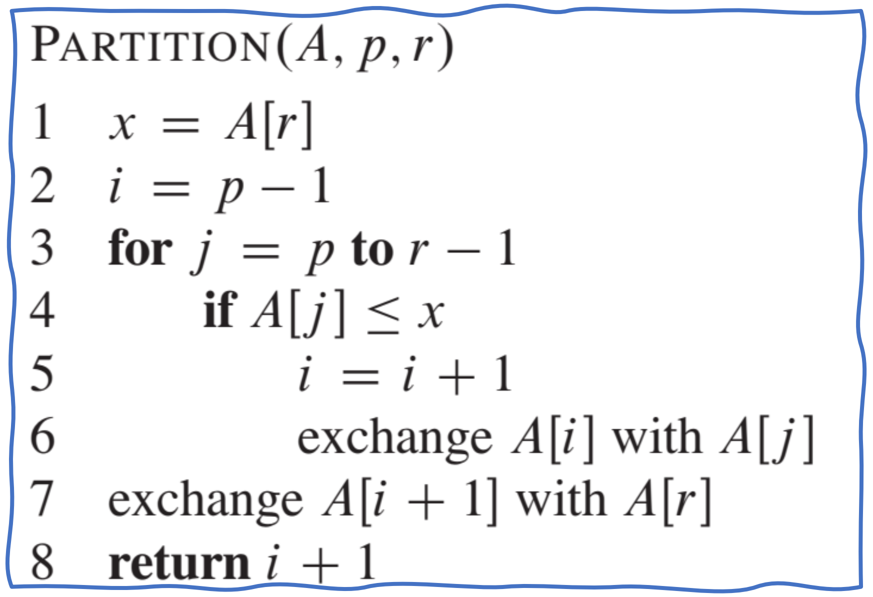
The new quicksort calls RANDOMIZED-PARTITION in place of PARTITION:

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

Expected Running Time of Quicksort

Before we jump to the running time analysis:

- Quicksort is dominated by the PARTITION procedure.
- Each time PARTITION procedure is called, a pivot element is selected.
- This element is never included in any future recursive call to Quick Sort.
- For n element in an array, there can be at most n calls to PARTITION procedure.
- Each call to partition takes $O(1)$ plus time that is proportional to the number of iteration in line 3-6.
  - Each iteration performs a comparison in line 4
  - If we can count **the total number of times that line 4 is executed <u>over the entire array</u>**, we can bound the total time spent in the for loop during the entire execution of Quick Sort.



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

Expected Running Time of Quick Sort

- Lemma:

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

- Now we are going to compute X, the total number of comparisons performed in **all calls** to PARTITION procedure.
  - By deriving an overall bound on the total number of comparisons. So we need to understand when the algorithm compares two elements in an array, when it does not.

Expected Running Time of Quick Sort

- For ease of analysis, we rename an array A as $z_1, z_2, z_3, \dots, z_n$ where z_i is the i-th smallest element in A.
    - Example: if $A = \{2,8,7,9,3,5,6,4\}$, then $z_1 = 2$, $z_2 = 3$, and so on

- Then we define $Z_{i,j} = \{z_i, z_{i+1}, z_{i+2}, \dots z_j\}$ to be multi set of elements between z_i and z_j (inclusive)
    - Example: if $A = \{2,8,7,9,3,5,6,4\}$, then $Z_{3,5} = \{4,5,6\}$

- Analyze when Quick Sort compares z_i and z_j **(assume that each element values are distinct)**.
    - First observe that each pair of elements is compared at most once **(Why?)**

- We need to evaluate the average number of comparisons performed within an entire execution of Quick Sort (calculate the $E[X]$)

Expected Running Time of Quick Sort

Define an indicator random variable for $X_{ij} = I\{z_i \text{ is compared to } z_j\} = \begin{cases} 1, if\ z_i \text{ is compared to } z_j \\ 0, otherwise \end{cases}$

Since each pair is compared at most once, the total number of comparisons performed by the algorithm is:
$X = (X_{1,2} + X_{1,3} + \cdots + X_{1,n}) + (X_{2,3} + X_{2,4} + \cdots + X_{2,n}) + \cdots + (X_{n,1} + \cdots + X_{n-1,n})$

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{i,j}$$

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

- Once a pivot x is chosen with $z_i < x < z_j$, then z_i and z_j cannot be compared at any subsequent time.
- z_i and z_j are compared if and only if the first element to be chosen as a pivot from Z_{ij} is either z_i or z_j

Expected Running Time of Quick Sort

- Illustration:
  - $A = \{2,8,7,9,3,5,6,4\}$
  - Suppose the first chosen pivot is 5, then A will be separated into $\{2,3,4\}$ and $\{6,7,8,9\}$.
  - The pivot element 5 is compared to all other elements, except for itself.
  - No number from the first set is compared to any from the second set.

  - In the set $Z_{1,4} = \{2,3,4,5\}$ → 2 is compared to 5 since the first chosen pivot is 5, but in the set $Z_{1,6} = \{2,3,4,5,6,7\}$, 2 is not compared to 7 because the first chosen pivot is neither $z_1 = 2$ nor $z_6 = 7$.

Expected Running Time of Quick Sort

- The whole set $Z_{i,j}$ is in the same partition.
- Therefore, any element of $Z_{i,j}$ is equally likely to be the first one chosen as a pivot.
- Because there are $j - i + 1$ elements in $Z_{i,j}$ and pivots are chosen randomly and independent, the probability of any element from $Z_{i,j}$ is chosen as the pivot is $\frac{1}{j-i+1}$

$$\begin{aligned}
\Pr\{z_i \text{ is compared to } z_j\} &= \Pr\{z_i \text{ or } z_j \text{ is first pivot chosen from } Z_{ij}\} \\
&= \Pr\{z_i \text{ is first pivot chosen from } Z_{ij}\} \\
&\qquad + \Pr\{z_j \text{ is first pivot chosen from } Z_{ij}\} \\
&= \frac{1}{j-i+1} + \frac{1}{j-i+1} \\
&= \frac{2}{j-i+1} .
\end{aligned}$$

Expected Running Time of Quick Sort

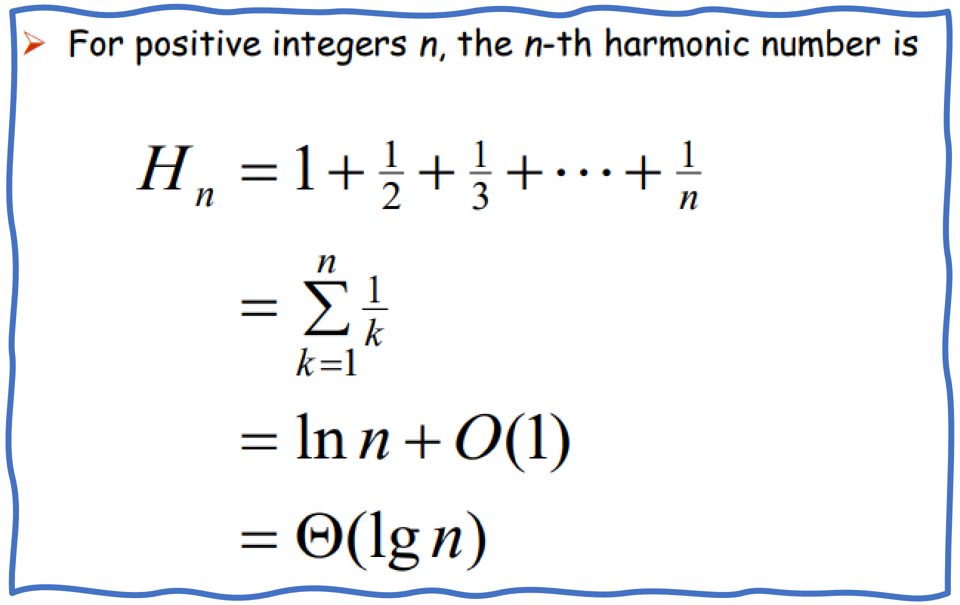
- $E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1}$
- Let $\mathrm{k} = j - i$, then

$$E[X] = \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} < \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k}$$

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

$$= O(n \lg n)$$

Harmonic series



➢ For positive integers *n*, the *n*-th harmonic number is

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$$

$$= \sum_{k=1}^{n} \frac{1}{k}$$

$$= \ln n + O(1)$$

$$= \Theta(\lg n)$$

Expected Running Time of Quick Sort

Conclusion

By using the RANDOMIZED-PARTITION, the expected running time is $\mathrm{O}(n \lg n)$ when element values are distinct.

Freivalds’ Algorithm

Matrix Multiplication Verification

- Given three matrices A, B, and C of size $n \times n$, is there a way to verify that C = A x B than actually computing A x B?
  - We cannot do sampling by choosing some entries from A and B then verify whether the multiplication result in C is correct. We need to check all the entries.
  - To do this checking, by standard matrix multiplication it needs $\Theta(n^3)$, while by Strassen algorithm (divide and conquer), it takes $\Theta(n^{2.81})$, slightly better than straightforward matrix multiplication.
  - Freivalds' algorithm reduces the running time to $\Theta(n^2)$ by utilizing randomization with high probability of its correctness.

Freivalds' Algorithm

- An example for **Monte Carlo** algorithm.
- Given three $n \times n$ matrices A, B, C, Freivalds' algorithm determines in $O(kn^2)$ whether the matrices are equal for a chosen k value with a probability of failure less than $\frac{1}{2^k}$.

Freivalds' Algorithm Idea

- Given $n \times n$ matrices A and B as the input.
- Choose a column vector $n \times 1$ $\vec{r} \in \{0,1\}^n$ uniformly and at random.
  - Each element of the vector is either 0 or 1.
- Compute $A(B\vec{r})$ and $C\vec{r}$.
  - This step takes $O(n^2)$ since it multiply an $n \times n$ matrix to a $n \times 1$ vector.
- The verification result:
  - If $A(B\vec{r}) \neq C\vec{r}$, the result is **false**. Thus, the multiplication result is incorrect.
  - If $A(B\vec{r}) = C\vec{r}$, the result is **true**. Thus, the multiplication result is correct.

Freivalds' Algorithm

- We run the algorithm k times. If $A\left(B\vec{r}^{\,i}\right) = C\vec{r}^{\,i}$ **for all** $i = 1,2,3,\dots,k$, then we can conclude that $A \times B = C$.
    - Since the matrix-vector multiplication takes $\mathrm{O}(n^2)$, then by k iterations, it takes $\mathrm{O}(kn^2)$.

- The error of the algorithm:
    - If $A \times B = C$, then the algorithm will always return true.
    - If $A \times B \neq C$, then the probability that the algorithm return true is $\leq \frac{1}{2}$. By iterating k times, the probability of error is $\leq \frac{1}{2^k}$

Freivalds’ Algorithm

Suppose one wished to determine whether:

$$AB = \begin{bmatrix} 2 & 3 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 1 & 2 \end{bmatrix} \overset{?}{=} \begin{bmatrix} 6 & 5 \\ 8 & 7 \end{bmatrix} = C.$$

A random two-element vector with entries equal to 0 or 1 is selected – say $\vec{r} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$ – and used to compute:

$$\begin{aligned} A \times (B\vec{r}) - C\vec{r} &= \begin{bmatrix} 2 & 3 \\ 3 & 4 \end{bmatrix} \left(\begin{bmatrix} 1 & 0 \\ 1 & 2 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} \right) - \begin{bmatrix} 6 & 5 \\ 8 & 7 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} \\ &= \begin{bmatrix} 2 & 3 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} 1 \\ 3 \end{bmatrix} - \begin{bmatrix} 11 \\ 15 \end{bmatrix} \\ &= \begin{bmatrix} 11 \\ 15 \end{bmatrix} - \begin{bmatrix} 11 \\ 15 \end{bmatrix} \\ &= \begin{bmatrix} 0 \\ 0 \end{bmatrix}. \end{aligned}$$

This yields the zero vector, suggesting the possibility that AB = C. However, if in a second trial the vector $\vec{r} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$ is selected, the result becomes:

$$A \times (B\vec{r}) - C\vec{r} = \begin{bmatrix} 2 & 3 \\ 3 & 4 \end{bmatrix} \left(\begin{bmatrix} 1 & 0 \\ 1 & 2 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} \right) - \begin{bmatrix} 6 & 5 \\ 8 & 7 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} -1 \\ -1 \end{bmatrix}.$$

The result is nonzero, proving that in fact AB ≠ C.

References

- Lecturer Slides by Bapak L. Yohanes Stefanus
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. Introduction to Algorithms, Third Edition (3rd. ed.). The MIT Press.
- https://people.seas.harvard.edu/~cs125/fall16/lec18.pdf