UMC 201: Data Structures & Algorithms

Lecture 4: Quicksort; Randomized Quicksort

Indian Institute of Science

Recap

- In the last lecture, we introduced the heap data structure & used it to design the heapsort algorithm.
- Heapsort has a running time of $O(n \log n)$ and it sorts the input array *in place*.

Recap

- In the last lecture, we introduced the heap data structure & used it to design the heapsort algorithm.
- Heapsort has a running time of $O(n \log n)$ and it sorts the input array *in place*.
- We also used heaps to implement the priority queues data structure that has several applications.

Recap

- In the last lecture, we introduced the heap data structure & used it to design the heapsort algorithm.
- Heapsort has a running time of $O(n \log n)$ and it sorts the input array *in place*.
- We also used heaps to implement the priority queues data structure that has several applications.

- In today's lecture, we'll discuss the quicksort algorithm, which is quite fast in practice.

Why another sorting algorithm?

- The quicksort algorithm has a worst-case running time of $\Theta(n^2)$.
- Nevertheless, it is *remarkably efficient in practice*.

Why another sorting algorithm?

- The quicksort algorithm has a worst-case running time of $\Theta(n^2)$.
- Nevertheless, it is *remarkably efficient in practice*.

- Its *expected running time* is $\Theta(n \log n)$, and the constant factor hidden in the $\Theta(\cdot)$ notation is small.
- It also has the advantage of *sorting in place*.

Quicksort

Quicksort: The idea

- Quicksort, like merge sort, applies the *divide-and-conquer method*. Suppose the input is $A[p:r]$.

Quicksort: The idea

- Quicksort, like merge sort, applies the *divide-and-conquer method*. Suppose the input is $A[p:r]$.

  - Divide by *partitioning* the array $A[p:r]$ into two subarrays $A[p:q-1]$ and $A[q+1:r]$ such that the elements of $A[p:q-1]$ are $\leq$ the *pivot* $A[q]$ and the elements of $A[q+1:r]$ are $> A[q]$.

Quicksort: The idea

- Quicksort, like merge sort, applies the *divide-and-conquer method*. Suppose the input is $A[p:r]$.

  - Divide by *partitioning* the array $A[p:r]$ into two subarrays $A[p:q-1]$ and $A[q+1:r]$ such that the elements of $A[p:q-1]$ are $\leq$ the *pivot* $A[q]$ and the elements of $A[q+1:r]$ are $> A[q]$.
  - Conquer by calling quicksort recursively to sort the *left* and the *right* subarrays $A[p:q-1]$ and $A[q+1:r]$, respectively.

Quicksort: Pseudocode

```
QUICKSORT(A, p, r)
1  if p < r
2      // Partition the subarray around the pivot, which ends up in A[q].
3      q = PARTITION(A, p, r)
4      QUICKSORT(A, p, q − 1)  // recursively sort the low side
5      QUICKSORT(A, q + 1, r)  // recursively sort the high side
```

- To sort an array $A[1:n]$, call QUICKSORT$(A, 1, n)$.

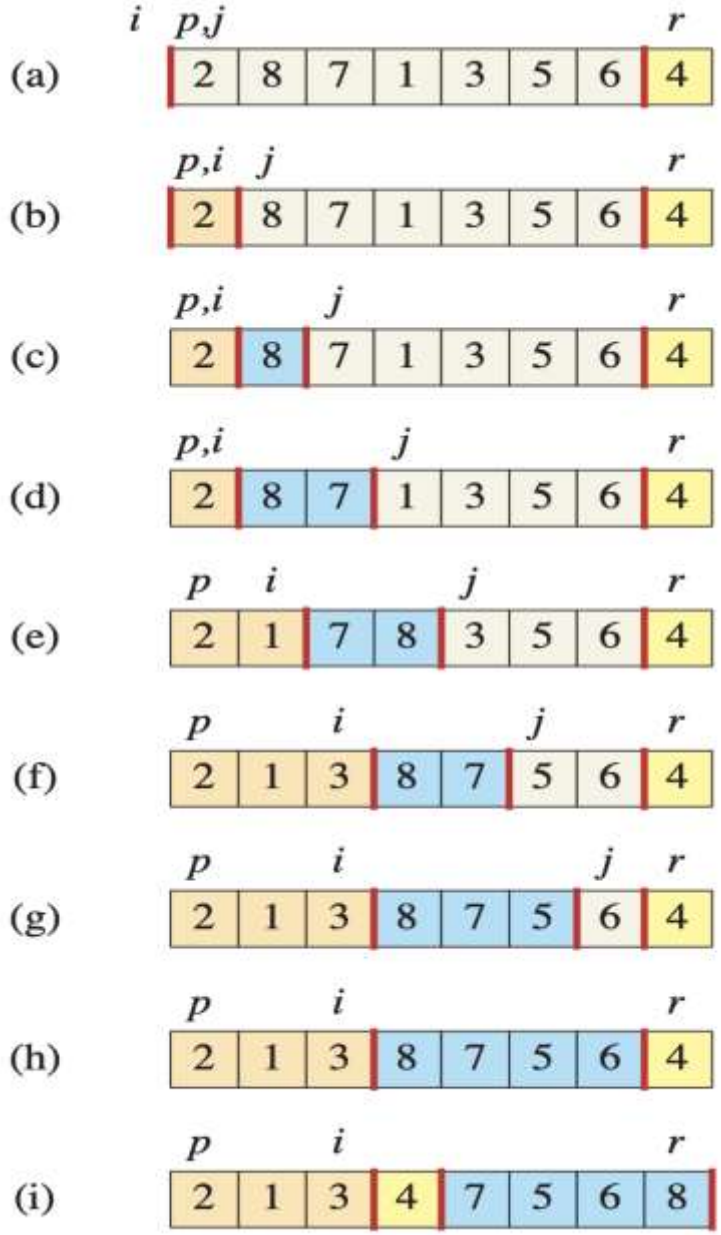
Quicksort: Pseudocode

```
QUICKSORT(A, p, r)
1  if p < r
2      // Partition the subarray around the pivot, which ends up in A[q].
3      q = PARTITION(A, p, r)
4      QUICKSORT(A, p, q − 1)  // recursively sort the low side
5      QUICKSORT(A, q + 1, r)  // recursively sort the high side
```

- To sort an array $A[1:n]$, call QUICKSORT$(A, 1, n)$.

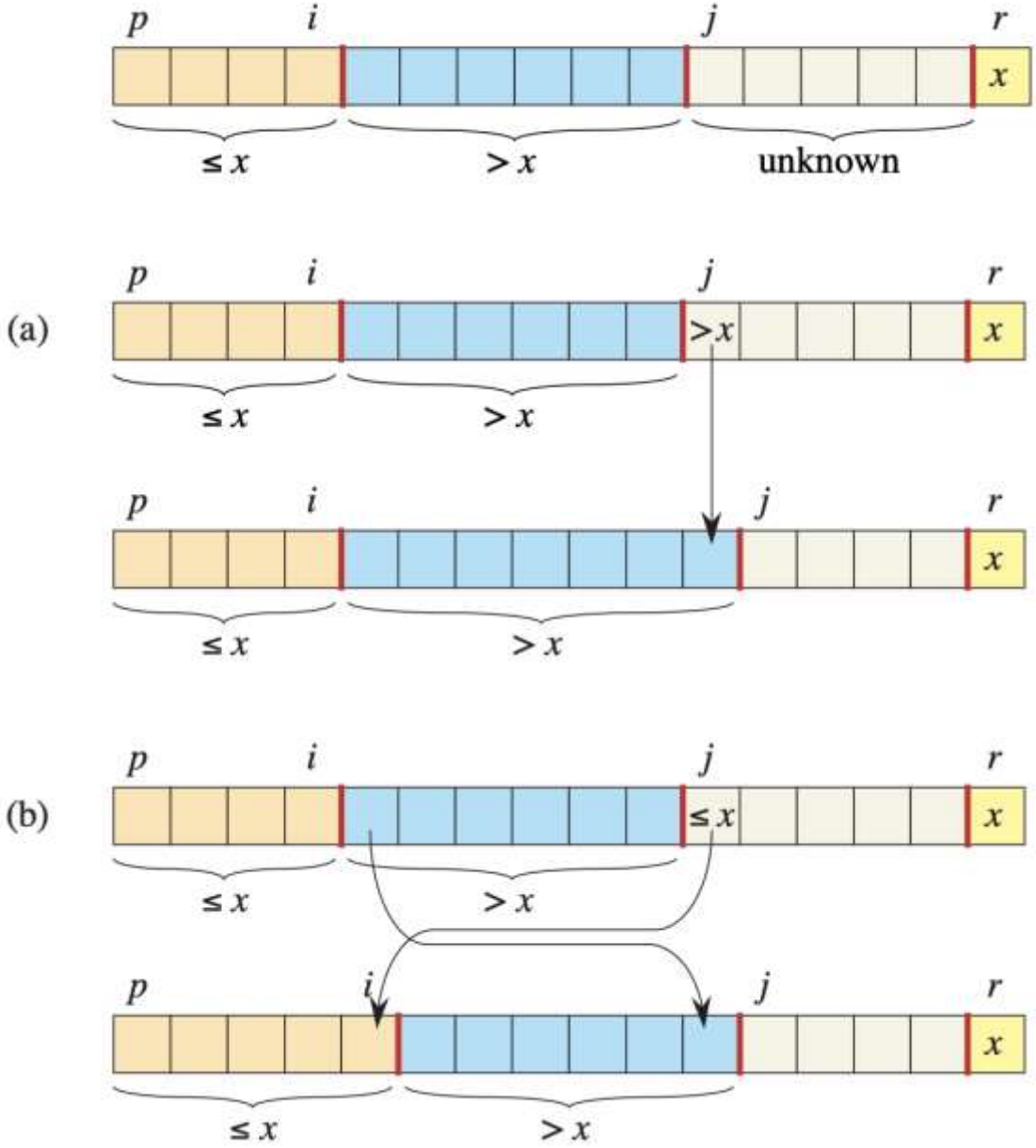
- How do we implement the PARTITION procedure?

Partition: An example



- PARTITION always selects the element $x = A[r]$ as the pivot. Here, 4 is the pivot.
- The indices i and j keep track of the left and right subarrays.

Partition: The strategy



Note. Index j moves forward in both Case (a) and (b), whereas index i moves forward only in Case (b) when an "exchange" happens.

Partition: Pseudocode

```
PARTITION(A, p, r)
1  x = A[r]                          // the pivot
2  i = p − 1                         // highest index into the low side
3  for j = p to r − 1                // process each element other than the pivot
4      if A[j] ≤ x                   // does this element belong on the low side?
5          i = i + 1                 // index of a new slot in the low side
6          exchange A[i] with A[j]   // put this element there
7  exchange A[i + 1] with A[r]       // pivot goes just to the right of the low side
8  return i + 1                      // new index of the pivot
```

- Correctness. Follows from the earlier discussion.

Partition: Time complexity

```
PARTITION(A, p, r)
1  x = A[r]                          // the pivot
2  i = p − 1                         // highest index into the low side
3  for j = p to r − 1                // process each element other than the pivot
4      if A[j] ≤ x                   // does this element belong on the low side?
5          i = i + 1                 // index of a new slot in the low side
6          exchange A[i] with A[j]   // put this element there
7  exchange A[i + 1] with A[r]       // pivot goes just to the right of the low side
8  return i + 1                      // new index of the pivot
```

- Time complexity. Time spent in each iteration of the **for**-loop (*Steps 3-6*) is $\Theta(1)$. So, the running time of PARTITION is $\Theta(r - p + 1)$ (i.e., linear in the size of the array $A[p:r]$).

Quicksort: Time complexity

```
QUICKSORT(A, p, r)
1  if p < r
2      // Partition the subarray around the pivot, which ends up in A[q].
3      q = PARTITION(A, p, r)
4      QUICKSORT(A, p, q − 1)  // recursively sort the low side
5      QUICKSORT(A, q + 1, r)  // recursively sort the high side
```

- The running time depends on *how balanced each partitioning is*, which depends on which elements are chosen as pivots. If the partitioning is balanced, then the algorithm is as fast as *merge sort*:

$$T(n) = 2T(n/2) + \Theta(n) = \Theta(n \log n)$$

Quicksort: Time complexity

```
QUICKSORT(A, p, r)
1  if p < r
2      // Partition the subarray around the pivot, which ends up in A[q].
3      q = PARTITION(A, p, r)
4      QUICKSORT(A, p, q − 1)  // recursively sort the low side
5      QUICKSORT(A, q + 1, r)  // recursively sort the high side
```

- The running time depends on *how balanced each partitioning is*, which depends on which elements are chosen as pivots. If the partitioning is balanced, then the algorithm is as fast as *merge sort*. If not, then it is as slow as *insertion sort:* $T(n) = T(n-1) + \Theta(n)$

$$= \Theta(n^2)$$

Quicksort: Time complexity

```
QUICKSORT(A, p, r)
1  if p < r
2      // Partition the subarray around the pivot, which ends up in A[q].
3      q = PARTITION(A, p, r)
4      QUICKSORT(A, p, q − 1)  // recursively sort the low side
5      QUICKSORT(A, q + 1, r)  // recursively sort the high side
```

- Worst-case running time.

$$T(n) = \max\{T(q) + T(n-1-q) : q \in [0, n-1]\} + \Theta(n)$$

Quicksort: Time complexity

```
QUICKSORT(A, p, r)
1  if p < r
2      // Partition the subarray around the pivot, which ends up in A[q].
3      q = PARTITION(A, p, r)
4      QUICKSORT(A, p, q − 1)  // recursively sort the low side
5      QUICKSORT(A, q + 1, r)  // recursively sort the high side
```

- Worst-case running time.

$$T(n) = \max\{T(q) + T(n-1-q) : q \in [0, n-1]\} + \Theta(n)$$

- We can solve the recurrence using the *substitution method*.

Solving the recurrence by substitution

$T(n) = \max\{T(q) + T(n-1-q) : q \in [0, n-1]\} + \Theta(n)$

- Assume, $T(m) \leq cm^2$ for all $m < n$ and a sufficiently large constant c. Need to show: $T(n) \leq cn^2$.

Solving the recurrence by substitution

$$T(n) = \max\{T(q) + T(n-1-q) : q \in [0, n-1]\} + \Theta(n)$$

- Assume, $T(m) \leq cm^2$ for all $m < n$ and a sufficiently large constant c. Need to show: $T(n) \leq cn^2$.
- Substituting the guess into the recurrence,

$$\begin{aligned} T(n) &= \max\{cq^2 + c(n-1-q)^2 : q \in [0, n-1]\} + \Theta(n) \\ &= c \cdot \max\{q^2 + (n-1-q)^2 : q \in [0, n-1]\} + \Theta(n) \\ &= c \cdot \max\{(n-1)^2 + 2q(q-(n-1)) : q \in [0, n-1]\} + \Theta(n) \end{aligned}$$

Solving the recurrence by substitution

$$T(n) = \max\{T(q) + T(n-1-q) : q \in [0, n-1]\} + \Theta(n)$$

- Assume, $T(m) \leq cm^2$ for all $m < n$ and a sufficiently large constant c. Need to show: $T(n) \leq cn^2$.
- Substituting the guess into the recurrence,

$$\begin{aligned} T(n) &= \max\{cq^2 + c(n-1-q)^2 : q \in [0, n-1]\} + \Theta(n) \\ &= c \cdot \max\{q^2 + (n-1-q)^2 : q \in [0, n-1]\} + \Theta(n) \\ &= c \cdot \max\{(n-1)^2 + 2q(q-(n-1)) : q \in [0, n-1]\} + \Theta(n) \end{aligned}$$

- As $q \leq n-1$ implies $2q(q-(n-1)) \leq 0$, we have $T(n) \leq c(n-1)^2 + \Theta(n) \leq cn^2$ (if c is sufficiently larger than the constant hiding in $\Theta(n)$).

Quicksort: Time complexity

```
QUICKSORT(A, p, r)
1  if p < r
2      // Partition the subarray around the pivot, which ends up in A[q].
3      q = PARTITION(A, p, r)
4      QUICKSORT(A, p, q − 1)  // recursively sort the low side
5      QUICKSORT(A, q + 1, r)  // recursively sort the high side
```

- Worst-case running time.

$$T(n) = \max\{T(q) + T(n-1-q) : q \in [0, n-1]\} + \Theta(n)$$
$$= O(n^2).$$

Quicksort: Time complexity

```
QUICKSORT(A, p, r)
1  if p < r
2      // Partition the subarray around the pivot, which ends up in A[q].
3      q = PARTITION(A, p, r)
4      QUICKSORT(A, p, q − 1)  // recursively sort the low side
5      QUICKSORT(A, q + 1, r)  // recursively sort the high side
```

- Worst-case running time.

$$T(n) = \max\{T(q) + T(n-1-q) : q \in [0, n-1]\} + \Theta(n)$$
$$= O(n^2).$$

- Turns out that the *expected runtime* of a *randomized version* of quicksort is $O(n \log n)$.

Quicksort: Space complexity

```
QUICKSORT(A, p, r)
1  if p < r
2      // Partition the subarray around the pivot, which ends up in A[q].
3      q = PARTITION(A, p, r)
4      QUICKSORT(A, p, q − 1)  // recursively sort the low side
5      QUICKSORT(A, q + 1, r)  // recursively sort the high side
```

- Although quicksort sorts the array in place, the amount of memory it uses (apart from the input array) could be $\Theta(n)$ *due to the recursion depth*.
- Note. The recursive Heapsort algorithm has *auxiliary space complexity* $O(\log n)$.

Randomized Quicksort

Randomized Qsort

- For quicksort, *randomization* yields a quick and fast practical algorithm.

- Many *software libraries* provide a randomized version of quicksort as their algorithm of choice for sorting large data sets.

- In Unix systems, quicksort is available as a *C standard library* function called qsort.

Randomized Qsort: The idea

- Instead of using $A[r]$ as the pivot, *randomly choose the pivot* from the subarray $A[p:r]$, i.e., each element in $A[p:r]$ has equal probability of being chosen.

Randomized Qsort: The idea

- Instead of using $A[r]$ as the pivot, *randomly choose the pivot* from the subarray $A[p:r]$, i.e., each element in $A[p:r]$ has equal probability of being chosen.

- *Exchange* the randomly chosen pivot with $A[r]$ before partitioning.

- As the pivot is randomly chosen, we expect the split of the input array to be *reasonably well balanced*.

Randomized Qsort: Pseudocode

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

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)
```

- Except the (random) choice of the pivot, everything remains same as the QUICKSORT procedure.

Randomized Qsort: Running time

- We'll analyze the *expected running time* of RANDOMIZED-QUICKSORT.
- If the expected run-time is upper bounded by T, then by applying *Markov's inequality*, we get that the probability of the run-time exceeding cT is $\leq c^{-1}$.

Randomized Qsort: Running time

- We'll analyze the *expected running time* of RANDOMIZED-QUICKSORT.
- If the expected run-time is upper bounded by T, then by applying *Markov's inequality*, we get that the probability of the run-time exceeding cT is $\leq c^{-1}$.

- Markov's inequality. If X is a *nonnegative* random variable and $a > 0$, then $\Pr(X \geq a) \leq \frac{E[X]}{a}$.

Randomized Qsort: Running time

- Lemma 1. The running time of QUICKSORT on an n-element array is $O(n + X)$, where X is the *number of comparisons* performed.

Randomized Qsort: Running time

- Lemma 1. The running time of QUICKSORT on an n-element array is $O(n + X)$, where X is the *number of comparisons* performed.
- *Proof.* Each time PARTITION is called, it selects a pivot, which is never included in future recursive calls to QUICKSORT and PARTITION.
- Thus, there can be at most n calls to PARTITION.

Randomized Qsort: Running time

```
PARTITION(A, p, r)
1  x = A[r]                          // the pivot
2  i = p − 1                         // highest index into the low side
3  for j = p to r − 1                // process each element other than the pivot
4      if A[j] ≤ x                   // does this element belong on the low side?
5          i = i + 1                 // index of a new slot in the low side
6          exchange A[i] with A[j]   // put this element there
7  exchange A[i + 1] with A[r]       // pivot goes just to the right of the low side
8  return i + 1                      // new index of the pivot
```

- *Proof (contd).* One call to PARTITION takes $O(1)$ time plus the amount of time that is proportional to the number of iterations of the **for** loop (*lines 3-6*).

Randomized Qsort: Running time

```
PARTITION(A, p, r)
1  x = A[r]                          // the pivot
2  i = p − 1                         // highest index into the low side
3  for j = p to r − 1                // process each element other than the pivot
4      if A[j] ≤ x                   // does this element belong on the low side?
5          i = i + 1                 // index of a new slot in the low side
6          exchange A[i] with A[j]   // put this element there
7  exchange A[i + 1] with A[r]       // pivot goes just to the right of the low side
8  return i + 1                      // new index of the pivot
```

- *Proof (contd).* One call to PARTITION takes $O(1)$ time plus the amount of time that is proportional to the number of iterations of the **for** loop (*lines 3-6*).
- Each iteration compares the pivot with another element *once* (*line 4*).

Randomized Qsort: Running time

```
PARTITION(A, p, r)
1  x = A[r]                          // the pivot
2  i = p − 1                         // highest index into the low side
3  for j = p to r − 1                // process each element other than the pivot
4      if A[j] ≤ x                   // does this element belong on the low side?
5          i = i + 1                 // index of a new slot in the low side
6          exchange A[i] with A[j]   // put this element there
7  exchange A[i + 1] with A[r]       // pivot goes just to the right of the low side
8  return i + 1                      // new index of the pivot
```

- *Proof (contd).* So, the total time spent in the **for** loop across all executions of PARTITION is proportional to X. And the total time spent in PARTITION outside the **for** loop is $O(n)$.

Randomized Qsort: Running time

- Lemma 1. The running time of QUICKSORT on an n-element array is $O(n + X)$, where X is the *number of comparisons* performed.
- *Proof.* Each time PARTITION is called, it selects a pivot, which is never included in future recursive calls to QUICKSORT and PARTITION.
- Thus, there can be at most n calls to PARTITION.

- Therefore, the total time for QUICKSORT is proportional to $n + X$.

Randomized Qsort: Running time

- Our goal for analysing the expected run-time of RANDOMIZED-QUICKSORT is to compute $E[X]$, where X denoted the total number of comparisons.

Randomized Qsort: Running time

- Our goal for analysing the expected run-time of RANDOMIZED-QUICKSORT is to compute $E[X]$, where X denoted the total number of comparisons.

- We'll refer to the elements of the array by $z_1, z_2, \ldots, z_n$, where $z_1 < z_2 < \cdots < z_n$.

- We denote the set $\{z_i, z_{i+1}, \ldots, z_j\}$ by Z_{ij}.

Randomized Qsort: Running time

- Lemma 2. During the execution of RANDOMIZED-QUICKSORT, an element z_i is compared with an element z_j, if and only if one of them is chosen as a pivot *before any other element in* Z_{ij}. Moreover, no two elements are ever compared twice.

Randomized Qsort: Running time

- Lemma 2. During the execution of RANDOMIZED-QUICKSORT, an element z_i is compared with an element z_j, if and only if one of them is chosen as a pivot *before any other element in* Z_{ij}. Moreover, no two elements are ever compared twice.
- *Proof.* As long as a pivot is **not** chosen from within Z_{ij}, the set is not "fragmented" by PARTITION.

Randomized Qsort: Running time

- Lemma 2. During the execution of RANDOMIZED-QUICKSORT, an element z_i is compared with an element z_j, if and only if one of them is chosen as a pivot *before any other element in* Z_{ij}. Moreover, no two elements are ever compared twice.
- *Proof.* As long as a pivot is **not** chosen from within Z_{ij}, the set is not "fragmented" by PARTITION.
- The first time a pivot x is chosen from Z_{ij}, if x is neither z_i nor z_j, then z_i and z_j fall into different sides of the partition and never compared later.

Randomized Qsort: Running time

- Lemma 2. During the execution of RANDOMIZED-QUICKSORT, an element z_i is compared with an element z_j, if and only if one of them is chosen as a pivot *before any other element in* Z_{ij}. Moreover, no two elements are ever compared twice.
- *Proof.* As long as a pivot is **not** chosen from within Z_{ij}, the set is not "fragmented" by PARTITION.
- If $x = z_i$ or z_j, then PARTITION compares it with every other element of Z_{ij} once.

Randomized Qsort: Running time

- Lemma 3. Consider an execution of the procedure RANDOMIZED-QUICKSORT on a array of n distinct element $z_1 < z_2 < \cdots < z_n$. Given two arbitrary elements z_i and z_j, where $i < j$, the probability that they are compared is $2/(j - i + 1)$.

Randomized Qsort: Running time

- Lemma 3. Consider an execution of the procedure RANDOMIZED-QUICKSORT on a array of n distinct element $z_1 < z_2 < \cdots < z_n$. Given two arbitrary elements z_i and z_j, where $i < j$, the probability that they are compared is $2/(j - i + 1)$.
- *Proof.* The elements of Z_{ij} all stay together until PARTITION chooses an element in Z_{ij} as pivot.

Randomized Qsort: Running time

- Lemma 3. Consider an execution of the procedure RANDOMIZED-QUICKSORT on a array of n distinct element $z_1 < z_2 < \cdots < z_n$. Given two arbitrary elements z_i and z_j, where $i < j$, the probability that they are compared is $2/(j - i + 1)$.
- *Proof.* The elements of Z_{ij} all stay together until PARTITION chooses an element in Z_{ij} as pivot.
- The first time PARTITION chooses a pivot x from Z_{ij}, every element of Z_{ij} is *equally likely* to be x as pivot is chosen uniformly at random.

Randomized Qsort: Running time

- Lemma 3. Consider an execution of the procedure RANDOMIZED-QUICKSORT on a array of n distinct element $z_1 < z_2 < \cdots < z_n$. Given two arbitrary elements z_i and z_j, where $i < j$, the probability that they are compared is $2/(j - i + 1)$.
- *Proof.* Therefore,

$$\Pr\{z_i \text{ is compared with } z_j\} = \frac{2}{j-i+1},$$

since $|Z_{ij}| = j - i + 1$.

Randomized Qsort: Running time

- Lemma 4. The expected running time of RANDOMIZED-QUICKSORT is $O(n \log n)$.

Randomized Qsort: Running time

- Lemma 4. The expected running time of RANDOMIZED-QUICKSORT is $O(n \log n)$.
- *Proof.* Let X_{ij} be the *indicator random variable*, which is 1 if z_i is compared with z_j, & 0 otherwise. Then, the total number of comparisons is $X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$

Randomized Qsort: Running time

- Lemma 4. The expected running time of RANDOMIZED-QUICKSORT is $O(n \log n)$.
- *Proof.* Let X_{ij} be the *indicator random variable*, which is 1 if z_i is compared with z_j, & 0 otherwise. Then, the total number of comparisons is $X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$
- By *linearity of expectation*, $\mathrm{E}[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathrm{E}[X_{ij}]$.

Randomized Qsort: Running time

- Lemma 4. The expected running time of RANDOMIZED-QUICKSORT is $O(n \log n)$.
- *Proof.* Let X_{ij} be the *indicator random variable*, which is 1 if z_i is compared with z_j, & 0 otherwise. Then, the total number of comparisons is $X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$
- By *linearity of expectation*, $\mathrm{E}[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathrm{E}[X_{ij}]$.
- By Lemma 3, $\mathrm{E}[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1}$.

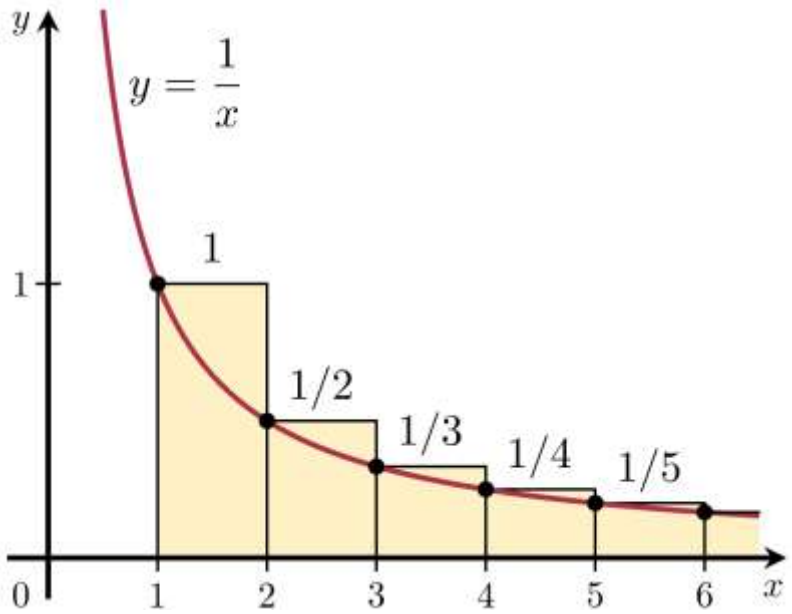
Randomized Qsort: Running time

- Lemma 4. The expected running time of RANDOMIZED-QUICKSORT is $O(n \log n)$.
- *Proof.* Let X_{ij} be the *indicator random variable*, which is 1 if z_i is compared with z_j, & 0 otherwise. Then, the total number of comparisons is $X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$
- By *linearity of expectation*, $\mathrm{E}[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathrm{E}[X_{ij}]$.
- By Lemma 3, $\mathrm{E}[X] = \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1}$.

Randomized Qsort: Running time

- Lemma 4. The expected running time of RANDOMIZED-QUICKSORT is $O(n \log n)$.
- *Proof.* Let X_{ij} be the *indicator random variable*, which is 1 if z_i is compared with z_j, & 0 otherwise. Then, the total number of comparisons is $X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$
- By *linearity of expectation*, $\mathrm{E}[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathrm{E}[X_{ij}]$.
- By Lemma 3, $\mathrm{E}[X] < \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k}$.

Harmonic series: Bounds



- Lower bound.

$$\sum_{1}^{n} \frac{1}{k} \geq \int_{1}^{n+1} \frac{dx}{x}$$

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

- Upper bound. $\sum_{k=1}^{n} \frac{1}{k} = \sum_{k=2}^{n} \frac{1}{k} + 1$

$$\leq \int_{1}^{n} \frac{dx}{x} + 1$$

$$= \ln n + 1$$

Randomized Qsort: Running time

- Lemma 4. The expected running time of RANDOMIZED-QUICKSORT is $O(n \log n)$.
- *Proof.* Let X_{ij} be the *indicator random variable*, which is 1 if z_i is compared with z_j, & 0 otherwise. Then, the total number of comparisons is $X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$
- By *linearity of expectation*, $\mathrm{E}[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathrm{E}[X_{ij}]$.
- By Lemma 3, $\mathrm{E}[X] < \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k}$

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

Practice problems

Practice Problem 1

Consider an array with distinct elements and for which all permutations of the elements are equally likely. Argue that for any constant $0 < \alpha \leq 1/2$, the probability is approximately $1 - 2\alpha$ that PARTITION produces a split at least as balanced as $1 - \alpha$ to α.

Practice Problem 2

```
HOARE-PARTITION(A, p, r)
 1  x = A[p]
 2  i = p − 1
 3  j = r + 1
 4  while TRUE
 5       repeat
 6            j = j − 1
 7       until A[j] ≤ x
 8       repeat
 9            i = i + 1
10       until A[i] ≥ x
11       if i < j
12            exchange A[i] with A[j]
13       else return j
```

Show that

When HOARE-PARTITION terminates, it returns a value j such that $p \leq j < r$.

Every element of $A[p : j]$ is less than or equal to every element of $A[j + 1 : r]$ when HOARE-PARTITION terminates.

Practice Problem 3

```
STOOGE-SORT(A, p, r)
1  if A[p] > A[r]
2      exchange A[p] with A[r]
3  if p + 1 < r
4      k = ⌊(r − p + 1)/3⌋          // round down
5      STOOGE-SORT(A, p, r − k)     // first two-thirds
6      STOOGE-SORT(A, p + k, r)     // last two-thirds
7      STOOGE-SORT(A, p, r − k)     // first two-thirds again
```

Argue that the call STOOGE-SORT$(A, 1, n)$ correctly sorts the array $A[1:n]$.

Give a recurrence for the worst-case running time of STOOGE-SORT and a tight asymptotic (Θ-notation) bound on the worst-case running time.

Compare the worst-case running time of STOOGE-SORT with that of insertion sort, merge sort, heapsort, and quicksort. Do the professors deserve tenure?