Randomised algorithms
COMP4500/7500
Advanced Algorithms & Data Structures

October 19, 2020

Overview

- Admin/reminders
- Probabilistic Analysis and randomised algorithms
- Quicksort
    - Partition
    - Randomised quicksort

Admin

Lectures

- Week 11 (this week): probabilistic analysis and randomised algorithms
- Week 12: revision. Overview/recap followed by questions

Admin

Lectures

- Week 11 (this week): probabilistic analysis and randomised algorithms
- Week 12: revision. Overview/recap followed by questions

Tutorials:

- Week 11 (this week): revision and assignment 2 help
- Week 12: open revision of all material
- Note: No tutorial specific to this week's material

Average case analysis

$$
\begin{aligned}
T_{\text{Worst-case}}(n) &= \max_{|x|=n} T(x) \\
T_{\text{Best-case}}(n) &= \min_{|x|=n} T(x) \\
T_{\text{Average-case}}(n) &= \sum_{|x|=n} T(x) \cdot Pr\{x\}
\end{aligned}
$$

Hire assistant example

HIRE-ASSISTANT(n)

$best = 0$ // candidate 0 is a least-qualified dummy candidate
for $j = 1$ **to** n
    interview candidate j (cost c_i)
    if candidate j is better than the best candidate
        $best = j$
        hire candidate j (cost c_h)

Hire assistant example

HIRE-ASSISTANT(n)

```
best = 0    // candidate 0 is a least-qualified dummy candidate
for j = 1 to n
        interview candidate j (cost c_i)
        if candidate j is better than the best candidate
                best = j
                hire candidate j (cost c_h)
```

n the total number of candidates

m the number of candidates hired

Hire assistant example

```
HIRE-ASSISTANT(n)
best = 0    // candidate 0 is a least-qualified dummy candidate
for j = 1 to n
    interview candidate j (cost c_i)
    if candidate j is better than the best candidate
        best = j
        hire candidate j (cost c_h)
```

n the total number of candidates

m the number of candidates hired

- Actual cost is $n\,c_i + m\,c_h$

Hire assistant example

HIRE-ASSISTANT(n)

$best = 0$ // candidate 0 is a least-qualified dummy candidate
for $j = 1$ **to** n
    interview candidate j (cost c_i)
    if candidate j is better than the best candidate
        $best = j$
        hire candidate j (cost c_h)

n the total number of candidates

m the number of candidates hired

- Actual cost is $n\,c_i + m\,c_h$
- Worst case is $n(c_i + c_h)$ is when $m = n$

Hire assistant example

HIRE-ASSISTANT(n)

```
best = 0    // candidate 0 is a least-qualified dummy candidate
for j = 1 to n
    interview candidate j (cost c_i)
    if candidate j is better than the best candidate
        best = j
        hire candidate j (cost c_h)
```

n the total number of candidates

m the number of candidates hired

- Actual cost is $n\,c_i + m\,c_h$
- Worst case is $n(c_i + c_h)$ is when $m = n$
- Average case is?

Hire assistant example: average-case analysis

Probability of hiring the j^{th} candidate

- Assume candidates are in random order
- Any of the first j candidates is equally likely to be the best

Hire assistant example: average-case analysis

Probability of hiring the j^{th} candidate

- Assume candidates are in random order
- Any of the first j candidates is equally likely to be the best
- The probability the j^{th} candidate is the best is:

Hire assistant example: average-case analysis

Probability of hiring the j^{th} candidate

- Assume candidates are in random order
- Any of the first j candidates is equally likely to be the best
- The probability the j^{th} candidate is the best is: $\frac{1}{j}$

Hire assistant example: average-case analysis

Probability of hiring the j^{th} candidate

- Assume candidates are in random order
- Any of the first j candidates is equally likely to be the best
- The probability the j^{th} candidate is the best is: $\frac{1}{j}$

Average cost of hiring candidates:

$$c_h \sum_{j=1}^{n} \frac{1}{j}$$

Hire assistant example: average-case analysis

Probability of hiring the j^{th} candidate

- Assume candidates are in random order
- Any of the first j candidates is equally likely to be the best
- The probability the j^{th} candidate is the best is: $\frac{1}{j}$

Average cost of hiring candidates:

$$c_h \sum_{j=1}^{n} \frac{1}{j} = c_h \ln n + O(1)$$

Hire assistant example: average-case analysis

Probability of hiring the j^{th} candidate

- Assume candidates are in random order
- Any of the first j candidates is equally likely to be the best
- The probability the j^{th} candidate is the best is: $\frac{1}{j}$

Average cost of hiring candidates:

$$c_h \sum_{j=1}^{n} \frac{1}{j} = c_h \ln n + O(1)$$

which is much better than the worst case of $c_h n$.

Hire assistant example: average-case analysis

Probability of hiring the j^{th} candidate

- Assume candidates are in random order
- Any of the first j candidates is equally likely to be the best
- The probability the j^{th} candidate is the best is: $\frac{1}{j}$

Average cost of hiring candidates:

$$c_h \sum_{j=1}^{n} \frac{1}{j} = c_h \ln n + O(1)$$

which is much better than the worst case of $c_h n$.

Recall the Harmonic series $\sum_{j=1}^{n} \frac{1}{j} = \ln n + O(1)$

Randomised Algorithms

An algorithm is **randomised** if its behaviour is determined by both:

- its inputs
- values produced by a random number generator

Randomised Algorithms

An algorithm is **randomised** if its behaviour is determined by both:

- its inputs
- values produced by a random number generator

For **deterministic (i.e. not randomised) algorithms** we can calculate the **average** running time, based on a probability distribution of inputs.

Randomised Algorithms

An algorithm is **randomised** if its behaviour is determined by both:

- its inputs
- values produced by a random number generator

For **deterministic (i.e. not randomised) algorithms** we can calculate the **average** running time, based on a probability distribution of inputs.

For **randomized algorithms** we calculate the **expected** running time – without having to make an assumption about the probability distribution of inputs.

Randomised hire assistant

RANDOMIZED-HIRE-ASSISTANT(n)

randomly permute the list of candidates
$best = 0$ // candidate 0 is a least-qualified dummy candidate
for $j = 1$ **to** n
    interview candidate j (cost c_i)
    if candidate j is better than the best candidate
        $best = j$
        hire candidate j (cost c_h)

Randomly permuting arrays: first method

We want a **uniform random permutation**:
$1/n!$ chance of each permutation of the n elements of array A

Randomly permuting arrays: first method

We want a **uniform random permutation**:
$1/n!$ chance of each permutation of the n elements of array A

PERMUTE-BY-SORT(A)

```
n = A.length
let P[1..n] be a new array
for i = 1 to n
        P[i] = RANDOM(1, n^3)
sort A, using P as the sort keys
```

Randomly permuting arrays: first method

We want a **uniform random permutation**:
$1/n!$ chance of each permutation of the n elements of array A

PERMUTE-BY-SORT(A)

```
n = A.length
let P[1..n] be a new array
for i = 1 to n
        P[i] = RANDOM(1, n^3)
sort A, using P as the sort keys
```

This algorithm produces a uniform random permutation if the chosen keys in P are unique:

Randomly permuting arrays: first method

We want a **uniform random permutation**:
$1/n!$ chance of each permutation of the n elements of array A

PERMUTE-BY-SORT(A)

```
n = A.length
let P[1..n] be a new array
for i = 1 to n
        P[i] = RANDOM(1, n^3)
sort A, using P as the sort keys
```

This algorithm produces a uniform random permutation if the chosen keys in P are unique:

$$\frac{n^3}{n^3} \times \frac{n^3 - 1}{n^3} \times \frac{n^3 - 2}{n^3} \times \cdots \times \frac{n^3 - n}{n^3}$$

Randomly permuting arrays: first method

We want a **uniform random permutation**:
$1/n!$ chance of each permutation of the n elements of array A

PERMUTE-BY-SORT(A)

```
n = A.length
let P[1..n] be a new array
for i = 1 to n
        P[i] = RANDOM(1, n^3)
sort A, using P as the sort keys
```

This algorithm produces a uniform random permutation if the chosen keys in P are unique:

$$\frac{n^3}{n^3} \times \frac{n^3-1}{n^3} \times \frac{n^3-2}{n^3} \times \cdots \times \frac{n^3-n}{n^3} \geq (1-\frac{1}{n^2})^n$$

Randomly permuting arrays: first method

We want a **uniform random permutation**:
$1/n!$ chance of each permutation of the n elements of array A

PERMUTE-BY-SORT(A)

$n = A.length$
let $P[1..n]$ be a new array
for $i = 1$ **to** n
    $P[i] = \text{RANDOM}(1, n^3)$
sort A, using P as the sort keys

This algorithm produces a uniform random permutation if the chosen keys in P are unique:

$$\frac{n^3}{n^3} \times \frac{n^3-1}{n^3} \times \frac{n^3-2}{n^3} \times \cdots \times \frac{n^3-n}{n^3} \geq (1 - \frac{1}{n^2})^n \geq 1 - \frac{1}{n}$$

Randomly permuting arrays: randomize in place

```
RANDOMIZE-IN-PLACE(A)
n = A.length
for i = 1 to n
        swap A[i] ↔ A[RANDOM(i, n)]
```

Randomly permuting arrays: randomize in place

RANDOMIZE-IN-PLACE(A)

$n = A.length$
for $i = 1$ **to** n
    swap $A[i] \leftrightarrow A[\textsc{Random}(i, n)]$

A k-permutation of set of n elements is defined to be a sequence containing k of the n elements.

Randomly permuting arrays: randomize in place

RANDOMIZE-IN-PLACE(A)

1 $n = A.length$
2 **for** $i = 1$ **to** n
3 swap $A[i] \leftrightarrow A[\text{RANDOM}(i, n)]$

A k-permutation of set of n elements is defined to be a sequence containing k of the n elements.

Invariant $Inv(i)$:
$A[1..i]$ contains any i-permutation of A with probability

$$\frac{(n-i)!}{n!}$$

Randomise-In-Place invariant initially

Before the first iteration $A[1..0]$ contains a 0-permutation with probability

$$1 = \frac{(n-0)!}{n!}$$

RANDOMISE-IN-PLACE maintains invariant

Assuming $Inv(i-1)$ holds before an iteration,
we must show $Inv(i)$ holds after.

RANDOMISE-IN-PLACE maintains invariant

Assuming $Inv(i-1)$ holds before an iteration,
we must show $Inv(i)$ holds after.

- Assume $A[1..i-1]$ contains any $(i-1)$-permutation with probability $\frac{(n-i+1)!}{n!}$.

RANDOMISE-IN-PLACE maintains invariant

Assuming $Inv(i-1)$ holds before an iteration,
we must show $Inv(i)$ holds after.

- Assume $A[1..i-1]$ contains any $(i-1)$-permutation with probability $\frac{(n-i+1)!}{n!}$.
- Consider the i-permutation $\langle x_1, \ldots, x_i \rangle$.

RANDOMISE-IN-PLACE maintains invariant

Assuming $Inv(i-1)$ holds before an iteration, we must show $Inv(i)$ holds after.

- Assume $A[1..i-1]$ contains any $(i-1)$-permutation with probability $\frac{(n-i+1)!}{n!}$.
- Consider the i-permutation $\langle x_1, \ldots, x_i \rangle$.
- Let E_1 be the event that $A[1..i-1]$ is $\langle x_1, \ldots, x_{i-1} \rangle$

RANDOMISE-IN-PLACE maintains invariant

Assuming $Inv(i-1)$ holds before an iteration,
we must show $Inv(i)$ holds after.

- Assume $A[1..i-1]$ contains any $(i-1)$-permutation with probability $\frac{(n-i+1)!}{n!}$.
- Consider the i-permutation $\langle x_1, \ldots, x_i \rangle$.
- Let E_1 be the event that $A[1..i-1]$ is $\langle x_1, \ldots, x_{i-1} \rangle$
- Let E_2 be the event that the i^{th} iteration places x_i in $A[i]$.

RANDOMISE-IN-PLACE maintains invariant

Assuming $Inv(i-1)$ holds before an iteration,
we must show $Inv(i)$ holds after.

- Assume $A[1..i-1]$ contains any $(i-1)$-permutation with probability $\frac{(n-i+1)!}{n!}$.
- Consider the i-permutation $\langle x_1, \ldots, x_i \rangle$.
- Let E_1 be the event that $A[1..i-1]$ is $\langle x_1, \ldots, x_{i-1} \rangle$
- Let E_2 be the event that the i^{th} iteration places x_i in $A[i]$.

$$Pr\{E_2 \cap E_1\} = Pr\{E_2 | E_1\} \cdot Pr\{E_1\}$$

RANDOMISE-IN-PLACE maintains invariant

Assuming $Inv(i-1)$ holds before an iteration,
we must show $Inv(i)$ holds after.

- Assume $A[1..i-1]$ contains any $(i-1)$-permutation with probability $\frac{(n-i+1)!}{n!}$.
- Consider the i-permutation $\langle x_1, \ldots, x_i \rangle$.
- Let E_1 be the event that $A[1..i-1]$ is $\langle x_1, \ldots, x_{i-1} \rangle$
- Let E_2 be the event that the i^{th} iteration places x_i in $A[i]$.

$$
\begin{aligned}
Pr\{E_2 \cap E_1\} &= Pr\{E_2 | E_1\} \cdot Pr\{E_1\} \\
&= \frac{1}{n-i+1} \cdot \frac{(n-i+1)!}{n!}
\end{aligned}
$$

RANDOMISE-IN-PLACE maintains invariant

Assuming $Inv(i-1)$ holds before an iteration,
we must show $Inv(i)$ holds after.

- Assume $A[1..i-1]$ contains any $(i-1)$-permutation with probability $\frac{(n-i+1)!}{n!}$.
- Consider the i-permutation $\langle x_1, \ldots, x_i \rangle$.
- Let E_1 be the event that $A[1..i-1]$ is $\langle x_1, \ldots, x_{i-1} \rangle$
- Let E_2 be the event that the i^{th} iteration places x_i in $A[i]$.

$$\begin{aligned}
Pr\{E_2 \cap E_1\} &= Pr\{E_2 | E_1\} \cdot Pr\{E_1\} \\
&= \frac{1}{n-i+1} \cdot \frac{(n-i+1)!}{n!} \\
&= \frac{(n-i)!}{n!}
\end{aligned}$$

RANDOMISE-IN-PLACE on termination

On termination $Inv(n)$ holds, i.e. $A[1..n]$ contains any n-permutation of the original array with probability

$$\frac{(n-n)!}{n!} = \frac{1}{n!}$$

RANDOMISE-IN-PLACE on termination

On termination $Inv(n)$ holds, i.e. $A[1..n]$ contains any n-permutation of the original array with probability

$$\frac{(n-n)!}{n!} = \frac{1}{n!}$$

Therefore A contains any permutation of the original array with probability

$$\frac{1}{n!}$$

RANDOMISE-IN-PLACE on termination

On termination $Inv(n)$ holds, i.e. $A[1..n]$ contains any n-permutation of the original array with probability

$$\frac{(n-n)!}{n!} = \frac{1}{n!}$$

Therefore A contains any permutation of the original array with probability

$$\frac{1}{n!}$$

Therefore each of the $n!$ possible permutations of the original array is equally likely.

Quicksort

1. Merge sort: divide & conquer
2. Heapsort: build and manipulate a heap
3. Quicksort: pre-process array by partitioning into elements greater-than and less-than some element (the “pivot”).

Quicksort

- Quicksort: pre-process data into “low” and “high” elements

  QUICKSORT(A, p, r)

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

Quicksort

- Quicksort: pre-process data into “low” and “high” elements

  QUICKSORT(A, p, r)

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

- Mergesort: post-process sorted data into a single, sorted array

  MERGE-SORT(A, p, r)

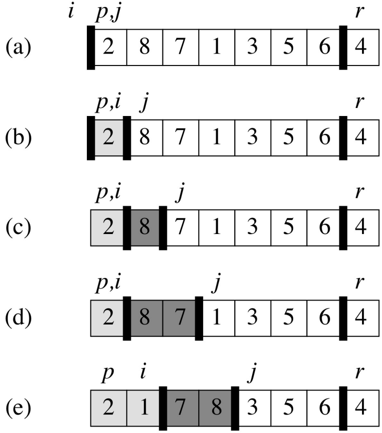
  if $p < r$
  $\quad q = \lfloor (p + r)/2 \rfloor$
  $\quad$ MERGE-SORT(A, p, q)
  $\quad$ MERGE-SORT($A, q + 1, r$)
  $\quad$ MERGE(A, p, q, r)

Quicksort: partition

PARTITION(A, p, r) rearranges A at the subrange $p..r$ (in place). Element $A[r]$ is the *pivot* value

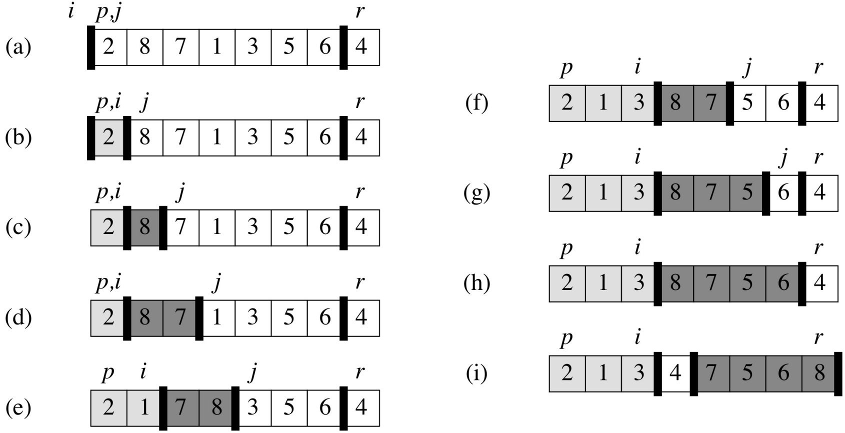
Quicksort: partition

PARTITION(A, p, r) rearranges A at the subrange $p..r$ (in place). Element $A[r]$ is the *pivot* value (4 in this case)



Quicksort: partition

PARTITION(A, p, r) rearranges A at the subrange $p..r$ (in place).
Element $A[r]$ is the *pivot* value (4 in this case)



Quicksort: partition

PARTITION(A, p, r)

```
x = A[r]
i = p - 1
for j = p to r - 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i+1
```

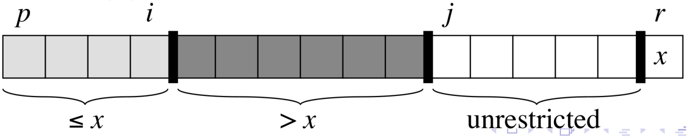
Partition is $\Theta(n)$.

Quicksort: partition

PARTITION(A, p, r)

```
x = A[r]
i = p − 1
for j = p to r − 1
        if A[j] ≤ x
                i = i + 1
                exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i+1
```

Partition is $\Theta(n)$.



Analysis of Quicksort

Performance depends on the element chosen as the pivot

- Best and average case: $\Theta(n \lg n)$.
- Worst case: $\Theta(n^2)$

Analysis of Quicksort

Performance depends on the element chosen as the pivot

- Best and average case: $\Theta(n \lg n)$.
- Worst case: $\Theta(n^2)$ = insertion sort.

Analysis of Quicksort

Performance depends on the element chosen as the pivot

- Best and average case: $\Theta(n \lg n)$.
- Worst case: $\Theta(n^2) =$ insertion sort.
  This case occurs if the array is already sorted. In fact in this special case, insertion sort is $\Theta(n)$.

Analysis of Quicksort (cont.)

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

1. Best case: $q = \lfloor (p + r)/2 \rfloor$
   $$T(n) = 2T(n/2) + \Theta(n)$$

Analysis of Quicksort (cont.)

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

1. Best case: $q = \lfloor (p+r)/2 \rfloor$
   $T(n) = 2T(n/2) + \Theta(n) \quad \in \Theta(n \lg n)$

Analysis of Quicksort (cont.)

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

1. Best case: $q = \lfloor (p + r)/2 \rfloor$
   $$T(n) = 2T(n/2) + \Theta(n) \quad \in \Theta(n \lg n)$$
2. Constant ratio split: $q = p + (r - p)/c$
   $$T(n) = T(n/c) + T((c-1)n/c) + \Theta(n)$$

Analysis of Quicksort (cont.)

$\textsc{Quicksort}(A, p, r)$

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

1. Best case: $q = \lfloor (p + r)/2 \rfloor$
   $$T(n) = 2T(n/2) + \Theta(n) \quad \in \Theta(n \lg n)$$
2. Constant ratio split: $q = p + (r - p)/c$
   $$T(n) = T(n/c) + T((c-1)n/c) + \Theta(n) \quad \in \Theta(n \lg n)$$

Analysis of Quicksort (cont.)

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

1. Best case: $q = \lfloor (p+r)/2 \rfloor$
   $$T(n) = 2T(n/2) + \Theta(n) \quad \in \Theta(n \lg n)$$
2. Constant ratio split: $q = p + (r-p)/c$
   $$T(n) = T(n/c) + T((c-1)n/c) + \Theta(n) \quad \in \Theta(n \lg n)$$
3. Worst case, no partitioning: $q = p$.
   $$T(n) = T(0) + T(n-1) + \Theta(n)$$

Analysis of Quicksort (cont.)

QUICKSORT(A, p, r)

1 **if** $p < r$
2 $\quad q = \text{PARTITION}(A, p, r)$
3 $\quad \text{QUICKSORT}(A, p, q-1)$
4 $\quad \text{QUICKSORT}(A, q+1, r)$

1. Best case: $q = \lfloor (p+r)/2 \rfloor$
   $$T(n) = 2T(n/2) + \Theta(n) \quad \in \Theta(n \lg n)$$
2. Constant ratio split: $q = p + (r-p)/c$
   $$T(n) = T(n/c) + T((c-1)n/c) + \Theta(n) \quad \in \Theta(n \lg n)$$
3. Worst case, no partitioning: $q = p$.
   $$T(n) = T(0) + T(n-1) + \Theta(n) \quad \in \Theta(n^2)$$

Analysis of QUICKSORT

- Consider the total number of comparisons of elements done by PARTITION over all calls by QUICKSORT

Analysis of QUICKSORT

- Consider the total number of comparisons of elements done by PARTITION over all calls by QUICKSORT
- Label the elements of A as $z_1, \ldots, z_n$,
  with z_i being the i^{th} smallest.

Analysis of QUICKSORT

- Consider the total number of comparisons of elements done by PARTITION over all calls by QUICKSORT
- Label the elements of A as $z_1, \ldots, z_n$, with z_i being the i^{th} smallest.
- Let $Z_{ij} = \{z_i, \ldots, z_j\}$

Analysis of QUICKSORT

- Consider the total number of comparisons of elements done by PARTITION over all calls by QUICKSORT
- Label the elements of A as $z_1, \ldots, z_n$, with z_i being the i^{th} smallest.
- Let $Z_{ij} = \{z_i, \ldots, z_j\}$

Analysis of QUICKSORT

- Consider an input array consisting of 1..10 in any order, and assume the first pivot is 4.

Analysis of QUICKSORT

- Consider an input array consisting of 1..10 in any order, and assume the first pivot is 4.
- The array is partitioned into:

$$\{1, 2, 3\} \text{ and } \{5, 6, 7, 8, 9, 10\}$$

Analysis of QUICKSORT

- Consider an input array consisting of 1..10 in any order, and assume the first pivot is 4.
- The array is partitioned into:

$$\{1, 2, 3\} \text{ and } \{5, 6, 7, 8, 9, 10\}$$

- In the partitioning:
  - the pivot 4 is compared with every other element

Analysis of QUICKSORT

- Consider an input array consisting of 1..10 in any order, and assume the first pivot is 4.
- The array is partitioned into:

$$\{1, 2, 3\} \text{ and } \{5, 6, 7, 8, 9, 10\}$$

- In the partitioning:
  - the pivot 4 is compared with every other element
  - but no element front he first set is or ever will be compared with an element of the second set

Probabiity z_i is compared with z_j

- For any elements z_i and z_j once a pivot x is chosen such that

$$z_i < x < z_j$$

  z_i and z_j can never be compared in the future.

Probabiity z_i is compared with z_j

- For any elements z_i and z_j once a pivot x is chosen such that

$$z_i < x < z_j$$

  z_i and z_j can never be compared in the future.
- If z_i is chosen as a pivot before any other element in Z_{ij} then z_i will be compared with every other element in Z_{ij}.

Probabiity z_i is compared with z_j

- For any elements z_i and z_j once a pivot x is chosen such that

$$z_i < x < z_j$$

  z_i and z_j can never be compared in the future.
- If z_i is chosen as a pivot before any other element in Z_{ij} then z_i will be compared with every other element in Z_{ij}. Similarly for z_j.

Probabiity z_i is compared with z_j

- For any elements z_i and z_j once a pivot x is chosen such that

$$z_i < x < z_j$$

  z_i and z_j can never be compared in the future.
- If z_i is chosen as a pivot before any other element in Z_{ij} then z_i will be compared with every other element in Z_{ij}. Similarly for z_j.
- Thus z_i and z_j are compared if and only if the first element chosen as a pivot in Z_{ij} is either z_i or z_j

Probabiity z_i is compared with z_j

- For any elements z_i and z_j once a pivot x is chosen such that

$$z_i < x < z_j$$

  z_i and z_j can never be compared in the future.
- If z_i is chosen as a pivot before any other element in Z_{ij} then z_i will be compared with every other element in Z_{ij}. Similarly for z_j.
- Thus z_i and z_j are compared if and only if the first element chosen as a pivot in Z_{ij} is either z_i or z_j
- Any element in Z_{ij} is equally likely and Z_{ij} has $j - i + 1$ elements

Analysis of QUICKSORT

Each pair of elements is compared at most once because in PARTITION elements are compared with the pivot only at most once, and an element is only used as a pivot in at most one call to PARTITION

Analysis of QUICKSORT

Each pair of elements is compared at most once because in PARTITION elements are compared with the pivot only at most once, and an element is only used as a pivot in at most one call to PARTITION

$$Pr\{z_i \text{ is compared with } z_j\}$$

Analysis of QUICKSORT

Each pair of elements is compared at most once because in PARTITION elements are compared with the pivot only at most once, and an element is only used as a pivot in at most one call to PARTITION

$$\begin{aligned} & Pr\{z_i \text{ is compared with } z_j\} \\ = \quad & Pr\{z_i \text{ or } z_j \text{ is the first pivot chosen from } Z_{ij}\} \end{aligned}$$

Analysis of QUICKSORT

Each pair of elements is compared at most once because in PARTITION elements are compared with the pivot only at most once, and an element is only used as a pivot in at most one call to PARTITION

$$
\begin{aligned}
& Pr\{z_i \text{ is compared with } z_j\} \\
= \; & Pr\{z_i \text{ or } z_j \text{ is the first pivot chosen from } Z_{ij}\} \\
= \; & Pr\{z_i \text{ is the first pivot chosen from } Z_{ij}\} + \\
& Pr\{z_j \text{ is the first pivot chosen from } Z_{ij}\}
\end{aligned}
$$

Analysis of QUICKSORT

Each pair of elements is compared at most once because in PARTITION elements are compared with the pivot only at most once, and an element is only used as a pivot in at most one call to PARTITION

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

Analysis of QUICKSORT

Each pair of elements is compared at most once because in PARTITION elements are compared with the pivot only at most once, and an element is only used as a pivot in at most one call to PARTITION

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

Number of comparisons over all calls to PARTITION

$$\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} Pr\{z_i \text{ is compared with } z_j\}$$

Number of comparisons over all calls to PARTITION

$$
\begin{aligned}
& \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} Pr\{z_i \text{ is compared with } z_j\} \\
= & \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1}
\end{aligned}
$$

Number of comparisons over all calls to PARTITION

$$
\begin{aligned}
& \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} Pr\{z_i \text{ is compared with } z_j\} \\
= & \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \\
= & \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1}, \quad \text{choosing } k = j - i
\end{aligned}
$$

Number of comparisons over all calls to PARTITION

$$
\begin{aligned}
& \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} Pr\{z_i \text{ is compared with } z_j\} \\
= & \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \\
= & \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1}, \quad \text{choosing } k = j - i \\
< & \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k}
\end{aligned}
$$

Number of comparisons over all calls to PARTITION

$$
\begin{aligned}
& \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} Pr\{z_i \text{ is compared with } z_j\} \\
= \; & \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \\
= \; & \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1}, \quad \text{choosing } k = j - i \\
< \; & \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k} \\
= \; & \sum_{i=1}^{n-1} O(\lg n), \quad \text{Harmonic series}
\end{aligned}
$$

Number of comparisons over all calls to PARTITION

$$
\begin{aligned}
& \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} Pr\{z_i \text{ is compared with } z_j\} \\
= & \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \\
= & \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1}, \quad \text{choosing } k = j - i \\
< & \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k} \\
= & \sum_{i=1}^{n-1} O(\lg n), \quad \text{Harmonic series} \\
= & \; O(n \lg n)
\end{aligned}
$$

Randomised Quicksort

- Despite the bad worst-case bound, Quicksort is regarded by many to be a good sorting algorithm.

Randomised Quicksort

- Despite the bad worst-case bound, Quicksort is regarded by many to be a good sorting algorithm.
- Good expected-case performance can be achieved by randomly permuting the input before sorting.
  (Permutation can be done in $\Theta(n)$)

Randomised Quicksort

- Despite the bad worst-case bound, Quicksort is regarded by many to be a good sorting algorithm.
- Good expected-case performance can be achieved by randomly permuting the input before sorting. (Permutation can be done in $\Theta(n)$)
- In practice, an even simpler approach specific to Quicksort is to choose a pivot from a random location. This just requires a constant-time swap of some random element $A[i]$ with $A[r]$.

Randomised Quicksort

- Despite the bad worst-case bound, Quicksort is regarded by many to be a good sorting algorithm.
- Good expected-case performance can be achieved by randomly permuting the input before sorting. (Permutation can be done in $\Theta(n)$)
- In practice, an even simpler approach specific to Quicksort is to choose a pivot from a random location. This just requires a constant-time swap of some random element $A[i]$ with $A[r]$.
- Having done this, the worst-case is unlikely, and the **expected time complexity** of the algorithm becomes $\Theta(n \lg n)$.

Recap

1. Randomised algorithms
2. Quicksort
    - Low overheads, sorts in-place, generally regarded as the fastest/most practical general sorting algorithm
    - Worst case is avoided by adding some randomness

Quick question

Given procedure BIASED-RANDOM() that returns 0 with probability p and 1 with probability $1-p$, where $0 < p < 1$,

Quick question

Given procedure BIASED-RANDOM() that returns 0 with probability p and 1 with probability $1-p$, where $0 < p < 1$, how can you implement procedure RANDOM that returns 0 or 1 with equal probability?

Quick question

RANDOM()

1 a = BIASED-RANDOM()
2 $\cdots$

- probability $a = 0$ is p
- probability $a = 1$ is $1-p$

Quick question

RANDOM()

1 $a = $ BIASED-RANDOM()
2 $b = $ BIASED-RANDOM()
3 $\cdots$

- probability $a = 0 \wedge b = 0$ is p^2
- probability $a = 0 \wedge b = 1$ is $p \times (1-p)$
- probability $a = 1 \wedge b = 0$ is $p \times (1-p)$
- probability $a = 1 \wedge b = 1$ is $(1-p)^2$

Quick question

RANDOM()

1 a = BIASED-RANDOM()
2 b = BIASED-RANDOM()
3 $\cdots$

- probability $a = 0 \wedge b = 0$ is p^2
- probability $a = 0 \wedge b = 1$ is $p \times (1-p)$
- probability $a = 1 \wedge b = 0$ is $p \times (1-p)$
- probability $a = 1 \wedge b = 1$ is $(1-p)^2$

So, the probability that $a = 0$ given that $a \neq b$ is:

$$\frac{p \times (1-p)}{p \times (1-p) + p \times (1-p)} = \frac{1}{2}$$

.

Quick question

RANDOM()

$a =$ BIASED-RANDOM()
$b =$ BIASED-RANDOM()
if $a \neq b$
    return a
else $\cdots$

Quick question

```
RANDOM()
a = BIASED-RANDOM()
b = BIASED-RANDOM()
while a = b
    a = BIASED-RANDOM()
    b = BIASED-RANDOM()
return a
```

Quick question

```
RANDOM()
a = BIASED-RANDOM()
b = BIASED-RANDOM()
while a = b
    a = BIASED-RANDOM()
    b = BIASED-RANDOM()
return a
```

What is the expected running time?

Quick question

Let $\alpha = 2(p \times (1-p))$ be the probability that $a \neq b$.

Quick question

Let $\alpha = 2(p \times (1-p))$ be the probability that $a \neq b$.

Probability of terminating after 0 loop iterations: α

Quick question

Let $\alpha = 2(p \times (1-p))$ be the probability that $a \neq b$.

Probability of terminating after 0 loop iterations: α
Probability of terminating after 1 loop iteration: $(1-\alpha) \times \alpha$

Quick question

Let $\alpha = 2(p \times (1-p))$ be the probability that $a \neq b$.

Probability of terminating after 0 loop iterations: α
Probability of terminating after 1 loop iteration: $(1-\alpha) \times \alpha$
Probability of terminating after 2 loop iteration: $(1-\alpha)^2 \times \alpha$

Quick question

Let $\alpha = 2(p \times (1-p))$ be the probability that $a \neq b$.

Probability of terminating after 0 loop iterations: α
Probability of terminating after 1 loop iteration: $(1-\alpha) \times \alpha$
Probability of terminating after 2 loop iteration: $(1-\alpha)^2 \times \alpha$
...
Probability of terminating after i loop iteration: $(1-\alpha)^i \times \alpha$

Quick question

Let $\alpha = 2(p \times (1-p))$ be the probability that $a \neq b$.

Probability of terminating after 0 loop iterations: α
Probability of terminating after 1 loop iteration: $(1-\alpha) \times \alpha$
Probability of terminating after 2 loop iteration: $(1-\alpha)^2 \times \alpha$
...
Probability of terminating after i loop iteration: $(1-\alpha)^i \times \alpha$

Expected number of loop iterations:

$$\sum_{i=0}^{\infty} i \times ((1-\alpha)^i \times \alpha)$$

Quick question

Let $\alpha = 2(p \times (1-p))$ be the probability that $a \neq b$.

Probability of terminating after 0 loop iterations: α
Probability of terminating after 1 loop iteration: $(1-\alpha) \times \alpha$
Probability of terminating after 2 loop iteration: $(1-\alpha)^2 \times \alpha$
...
Probability of terminating after i loop iteration: $(1-\alpha)^i \times \alpha$

Expected number of loop iterations:

$$\begin{aligned} & \sum_{i=0}^{\infty} i \times ((1-\alpha)^i \times \alpha) \\ = \; & \alpha \times (\sum_{i=0}^{\infty} i \times ((1-\alpha)^i)) \end{aligned}$$

Quick question

Let $\alpha = 2(p \times (1-p))$ be the probability that $a \neq b$.

Probability of terminating after 0 loop iterations: α
Probability of terminating after 1 loop iteration: $(1-\alpha) \times \alpha$
Probability of terminating after 2 loop iteration: $(1-\alpha)^2 \times \alpha$
...
Probability of terminating after i loop iteration: $(1-\alpha)^i \times \alpha$

Expected number of loop iterations:

$$
\begin{aligned}
& \textstyle\sum_{i=0}^{\infty} i \times ((1-\alpha)^i \times \alpha) \\
= \; & \alpha \times (\textstyle\sum_{i=0}^{\infty} i \times ((1-\alpha)^i)) \\
= \; & \alpha \times \frac{1-\alpha}{(1-(1-\alpha))^2} \\
= \; & \alpha \times \frac{1-\alpha}{\alpha^2} \\
= \; & \frac{1-\alpha}{\alpha} \\
= \; & \frac{1}{\alpha} - 1
\end{aligned}
$$