Analysis of Algorithms, I

CSOR W4231

Eleni Drinea
Computer Science Department

Columbia University

Quicksort, randomized quicksort, occupancy problems

Outline

Today

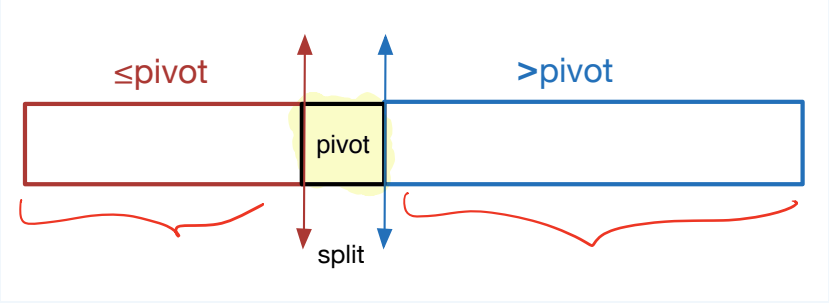
Quicksort facts

- Quicksort is a **divide and conquer** algorithm
- It is the standard algorithm used for sorting
- It is an **in-place** algorithm
- Its worst-case running time is $\Theta(n^2)$ but its average-case running time is $\Theta(n \log n)$
- We will use it to introduce **randomized** algorithms

Recursively Sort

Quicksort: main idea

- Pick an input item, call it *pivot*, and place it in its final location in the sorted array by re-organizing the array so that:
  - all items $\leq$ *pivot* are placed before *pivot*
  - all items $>$ *pivot* are placed after *pivot*



- Recursively sort the subarray to the left of *pivot*.
- Recursively sort the subarray to the right of *pivot*.

By doing so, you have placed the pivot item in its final location in the sorted array. — so you can continue to sort the arrays to the

Quicksort pseudocode

left and right of pivot.

Partition( ) called exactly n times because every item has to be chosen as a pivot item @ some point.

```
Quicksort(A, left, right)
    if |A| = 0 then return          //A is empty
    end if
    split = Partition(A, left, right)
    Quicksort(A, left, split − 1)
    Quicksort(A, split + 1, right)
```

← Returns the position of the pivot item in the reorganized array.

Initial call: `Quicksort`$(A, 1, n)$

left = all items in array less than pivot item

right = all items in array greater than pivot item

Where does all the work in Quicksort happen?

→ Partition(·)

Quicksort pseudocode

$T(n)$ = time for QS on input size n.

BEST CASE

$T(n) = cn + 2T(\frac{n}{2})$

$= O(n \log n)$

```
Quicksort(A, left, right)
    if |A| = 0 then return          //A is empty
    end if
    split = Partition(A, left, right)
    Quicksort(A, left, split − 1)
    Quicksort(A, split + 1, right)
```

(left = 1, right = n)

WORST CASE

$T(n) = cn + T(n-1)$

$= O(n^2)$

(Worst Case is for reverse order sorted input.)

Initial call: `Quicksort`$(A, 1, n)$

- Partition on input size n: $O(n)$
- n Partition calls:

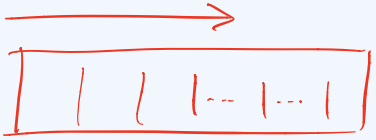
$\Rightarrow O(n^2)$

Subroutine `Partition`$(A, left, right)$

Notation: $A[i, j]$ denotes the portion of A starting at position i and ending at position j.

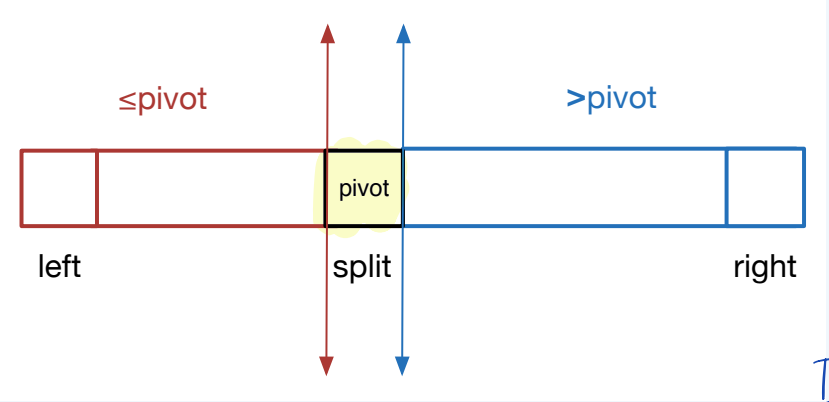
`Partition`$(A, left, right)$

1. picks a *pivot* item
2. re-organizes $A[left, right]$ so that
   - all items before *pivot* are $\leq$ *pivot*
   - all items after *pivot* are $>$ *pivot*
3. returns *split*, the index of *pivot* in the re-organized array

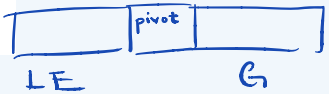


Scans through array and compares to pivot item.

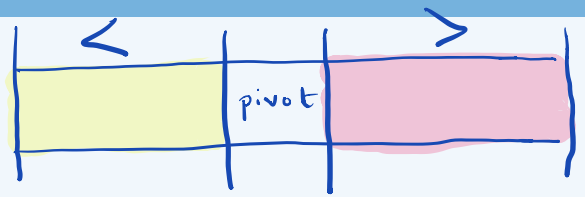
After `Partition`, $A[left, right]$ looks as follows:



Will have two lists:
one list w/ items ≤ pivot (LE)
and another > pivot (G)



Implementing `Partition`



How can we accomplish this?

1. Pick a *pivot* item: for simplicity, always pick the last item of the array as *pivot*, i.e., $pivot = A[right]$.
   - Thus $A[right]$ will be placed in its final location in the sorted output when `Partition` returns; it will never be used (or moved) again until the algorithm terminates.

2. Re-organize the input array A in place. *How?*

(What if we didn't care to implement `Partition` *in place?)*

Maintain two new lists:

1) one list for all items less than or equal to pivot (LE)

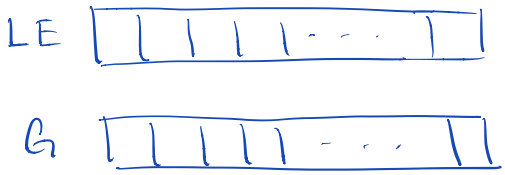
2) another list for all items greater than pivot. (G)

Non-in-place implementation:

Maintain two new lists:

one list for all items less than or equal to pivot (LE)

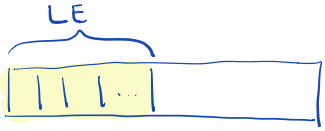
another list for all items greater than pivot. (G)



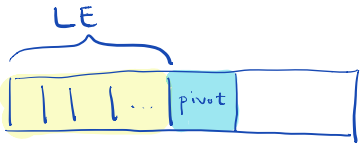
{ Requires $\Theta(n)$ extra space

Step 1: Place all elements $\leq$ pivot in LE, and all elements $>$ pivot in G.

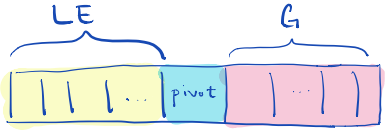
Step 2: Copy all elements in the LE list to an array.



Step 3: Copy pivot to the subsequent position in array



Step 4: Copy all elements in the G list to positions <u>after</u> pivot.

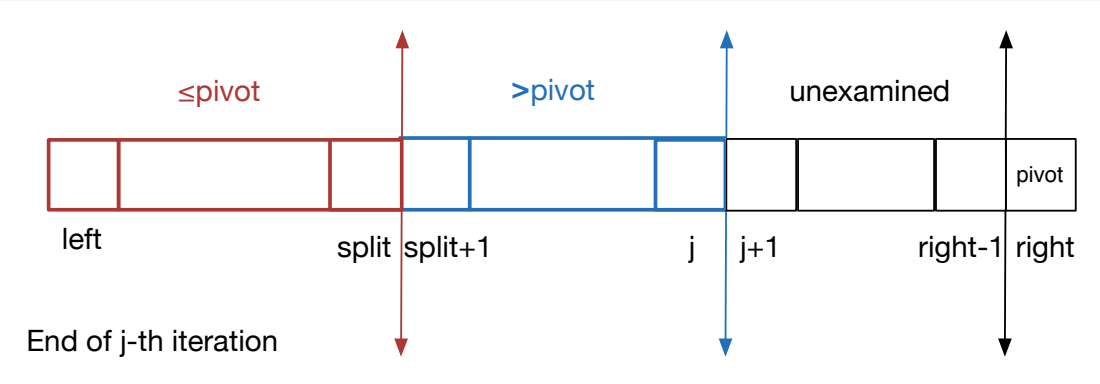


Steps 2-4 takes $O(n)$ time. – we are just scanning the items in the array and adding them to these lists, and then copying them to the output.

Implementing `Partition` in place

`Partition` examines the items in $A[left, right]$ one by one and maintains three regions in A. Specifically, **after** examining the j-th item for $j \in [left, right-1]$, the regions are:

1. Left region: starts at $left$ and ends at $split$;
   $A[left, split]$ contains all items $\leq pivot$ examined so far.
2. Middle region: starts at $split+1$ and ends at j;
   $A[split+1, j]$ contains all items $> pivot$ examined so far.
3. Right region: starts at $j+1$ and ends at $right-1$;
   $A[j+1, right-1]$ contains all unexamined items.

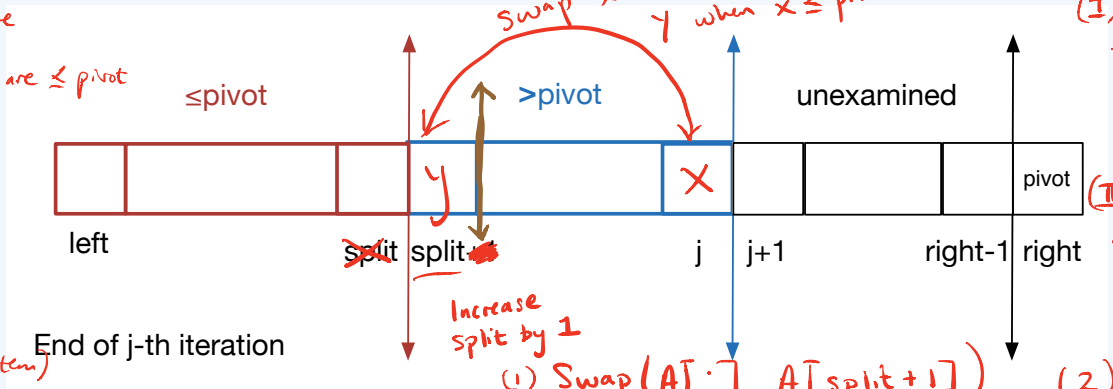


Implementing `Partition` in place

`Partition` examines the items in $A[left, right]$ one by one and maintains three regions in A. Specifically, **after** examining the j-th item for $j \in [left, right-1]$, the regions are:

1. Left region: starts at $left$ and ends at $split$;
   $A[left, split]$ contains all items $\leq pivot$ examined so far.
2. Middle region: starts at $split + 1$ and ends at j;
   $A[split + 1, j]$ contains all items $> pivot$ examined so far.
3. Right region: starts at $j + 1$ and ends at $right - 1$;
   $A[j + 1, right - 1]$ contains all unexamined items.

How can we make sure that all items from left to split are ≤ pivot and all items btwn split+1 to j are > pivot? (After examining jth item)



Two cases

(I) $x > pivot$
Don't need to do anything – already in correct location.

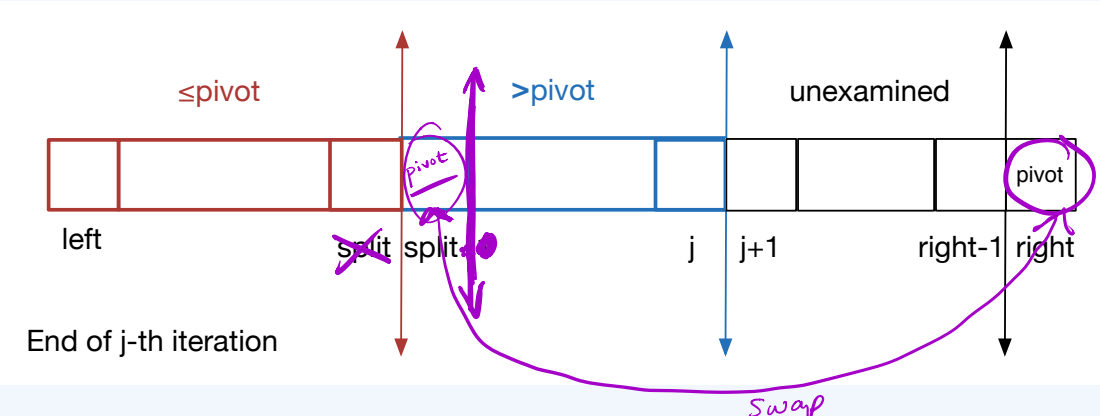
(II) $x \leq pivot$
- Swap x & y
- Increase split by one

(1) Swap(A[·], A[split+1]), (2) Split++

Implementing `Partition` in place

`Partition` examines the items in $A[left, right]$ one by one and maintains three regions in A. Specifically, **after** examining the j-th item for $j \in [left, right-1]$, the regions are:

1. Left region: starts at $left$ and ends at $split$;
   $A[left, split]$ contains all items $\leq pivot$ examined so far.
2. Middle region: starts at $split+1$ and ends at j;
   $A[split+1, j]$ contains all items $> pivot$ examined so far.
3. Right region: starts at $j+1$ and ends at $right-1$;
   $A[j+1, right-1]$ contains all unexamined items.



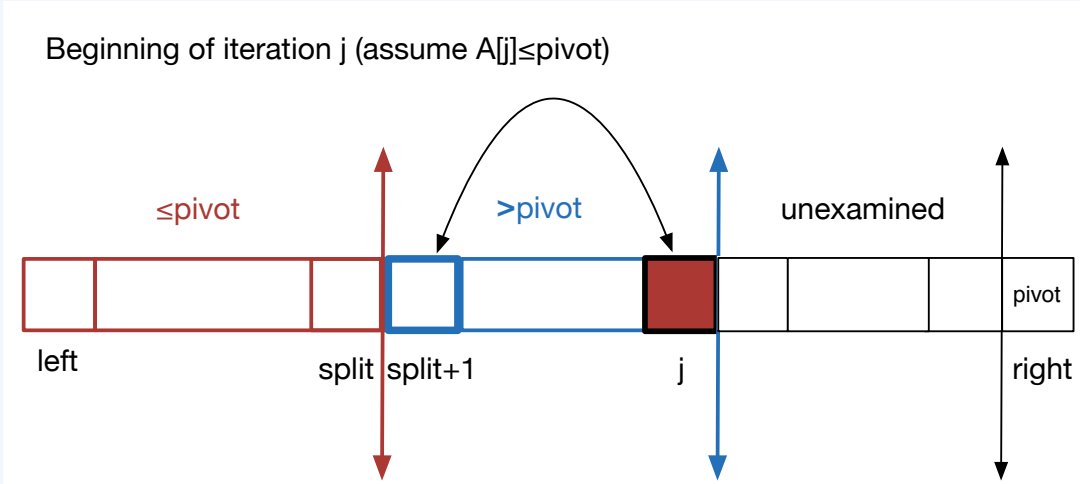
Implementing `Partition` in place (cont'd)

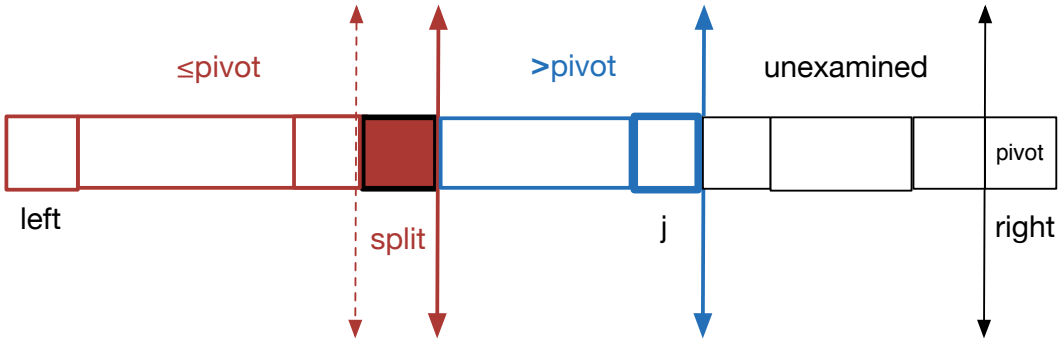
At the **beginning** of iteration j, $A[j]$ is compared with $pivot$.

If $A[j] \leq pivot$

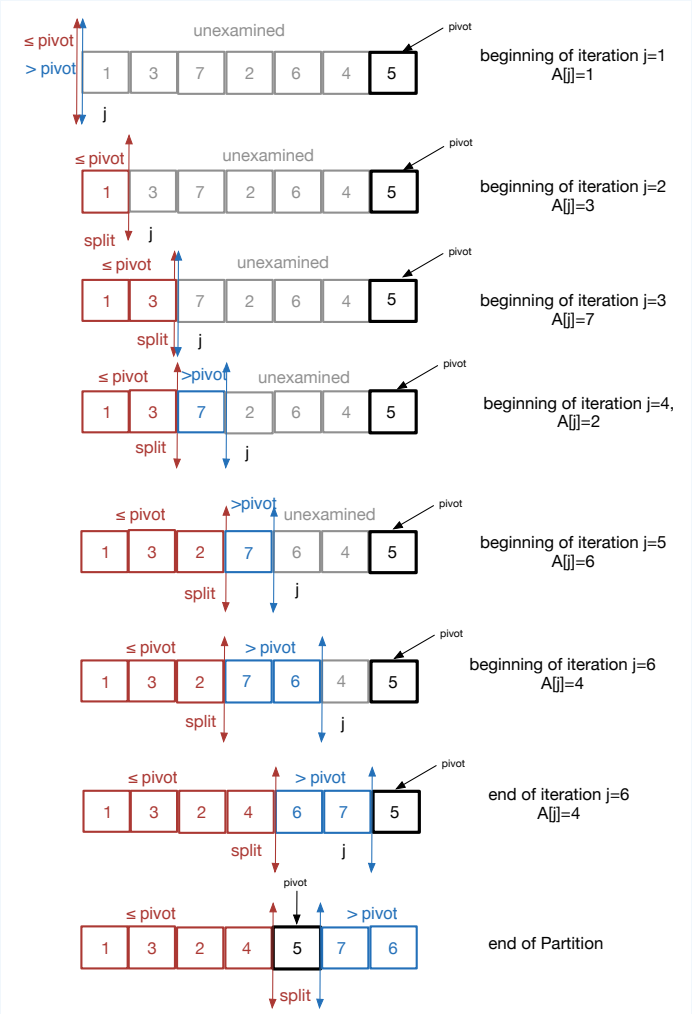
1. swap $A[j]$ with $A[split + 1]$, the first element of the middle region (items $> pivot$): since $A[split + 1] > pivot$, it is "safe" to move it to the end of the middle region
2. increment $split$ to include $A[j]$ in the left region (items $> pivot$)

Iteration j: when $A[j] \leq pivot$





Example: $A = \{1, 3, 7, 2, 6, 4, 5\}$, `Partition`$(A, 1, 7)$



Pseudocode for Partition

In Place Implementation

– No extra space needed

```
Partition(A, left, right)
O(1) → pivot = A[right]
O(1) → split = left − 1
       for j = left to right − 1 do        (j = 1 to n−1) → n−1 iterations
           if A[j] ≤ pivot then
               swap(A[j], A[split + 1])
               split = split + 1
           end if
       end for
O(1) → swap(pivot, A[split + 1])    //place pivot after A[split] (why?)
O(1) → return split + 1             //the final position of pivot
```

O(1) ⇒ for each iteration.

O(n) for the for-loop

⇒ Running time of Partition: O(n)

Analysis of `Partition`: correctness

Notation: $A[i, j]$ denotes the portion of A that starts at position i and ends at position j.

Claim 1.

For $left \leq j \leq right - 1$, at the end of loop j,

1. *all items in $A[left, split]$ are $\leq pivot$; and*
2. *all items in $A[split + 1, j]$ are $> pivot$*

Remark: If the claim is true, correctness of `Partition` follows *(why?)*.

Prove correctness by induction.

Claim 1.

For $left \le j \le right - 1$, at the end of loop j,

1. *all items in $A[left, split]$ are $\le pivot$; and*
2. *all items in $A[split + 1, j]$ are $> pivot$*

Base : $j = l$ $\quad$ $s = l - 1$ $\quad$ $l \equiv left$, $s \equiv split$

At the end of loop l,

1. $A[l, s]$ contains items $\le$ pivot
2. $A[s+1, l]$ contains items $>$ pivot

If $A[l] >$ pivot :

True → 1. $A[l, l-1]$ contains items $\le$ pivot ? ✓

True → 2. $A[l, l]$ contains items $>$ pivot ? ✓

If $A[l] \le$ pivot :

True → 1. $A[l, l]$ contains items $\le$ pivot ? ✓

True → 2. $A[l+1, l]$ contains items $>$ pivot ? ✓

Hence, the base case holds.

Hyp : Assume claim holds for all $j \in [l, r-1]$

Step : See Proof on following slides.

Proof of Claim 1

By induction on j.

1. **Base case:** For $j = left$ (that is, during the first execution of the for loop), there are two possibilities:
   - if $A[left] \leq pivot$, then $A[left]$ is swapped with itself and $split$ is incremented to equal $left$;
   - otherwise, nothing happens.

   In both cases, the claim holds for $j = left$.

2. **Hypothesis:** Assume that the claim is true for some $left \leq j < right - 1$.
   - That is, at the end of loop j, all items in $A[left, split]$ are $\leq pivot$ and all items in $A[split + 1, j]$ are $> pivot$.

Proof of Claim 1 (cont'd)

3. **Step:** We will show the claim for $j+1$. That is, we will show that after loop $j+1$, all items in $A[left, split]$ are $\leq pivot$ and all items in $A[split+1, j+1]$ are $> pivot$.

- At the beginning of loop $j+1$, by the hypothesis, items in $A[left, split]$ are $\leq pivot$ and items in $A[split+1, j]$ are $> pivot$.
- Inside loop $j+1$, there are two possibilities:
  1. $A[j+1] \leq pivot$: then $A[j+1]$ is swapped with $A[split+1]$. At this point, items in $A[left, split+1]$ are $\leq pivot$ and items in $A[split+2, j+1]$ are $> pivot$. Incrementing $split$ (the next step in the pseudocode) yields that the claim holds for $j+1$.
  2. $A[j+1] > pivot$: nothing is done. The truth of the claim follows from the hypothesis.

This completes the proof of the inductive step.

Analysis of `Partition`: running time and space

- **Running time:** on input size n, `Partition` goes through each of the $n - 1$ leftmost elements once and performs constant amount of work per element.

  $\Rightarrow$ `Partition` requires $\Theta(n)$ time.

- **Space:** in-place algorithm

Analysis of `Quicksort`: correctness

- `Quicksort` is a recursive algorithm; we will prove correctness by induction on the input size n.
- We will use **strong** induction: the induction step at n requires that the inductive hypothesis holds at all steps $1, 2, \ldots, n-1$ and not just at step $n-1$, as with simple induction.
- Strong induction is most useful when several instances of the hypothesis are required to show the inductive step.

Analysis of `Quicksort`: correctness

- **Base case:** for $n = 0$, `Quicksort` sorts correctly.
- **Hypothesis: for all** $0 \leq m < n$, `Quicksort` correctly sorts on input size m.
- **Step:** show that `Quicksort` correctly sorts on input size n.
  - `Partition`$(A, 1, n)$ re-organizes A so that all items
    - in $A[1, \ldots, split - 1]$ are $\leq A[split]$;
    - in $A[split + 1, \ldots, n]$ are $> A[split]$.
  - Next, `Quicksort`$(A, 1, split - 1)$, `Quicksort`$(A, split + 1, n)$ will correctly sort their inputs (by the hypothesis). Hence

$$A[1] \leq \ldots \leq A[split - 1] \text{ and } A[split + 1] \leq \ldots \leq A[n].$$

At this point, `Quicksort` terminates and A is sorted.

Analysis of `Quicksort`: space and running time

- **Space**: in-place algorithm
- **Running time** $T(n)$: depends on the arrangement of the input elements
  - the sizes of the inputs to the two recursive calls –hence the form of the recurrence– depend on how *pivot* compares to the rest of the input items

Running time of `Quicksort`: Best Case

Suppose that in every call to `Partition` the pivot item is the median of the input.

Then every `Partition` splits its input into two lists of almost equal sizes, thus

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

This is a "balanced" partitioning.

- Example of best case: $A = [1 \quad 3 \quad 2 \quad 5 \quad 7 \quad 6 \quad 4]$

Remark 1.

You can show that $T(n) = O(n \log n)$ for any splitting where the two subarrays have sizes αn, $(1 - \alpha)n$ respectively, for constant $0 < \alpha < 1$.

Running time of `Quicksort`: Worst Case

- Upper bound for worst-case running time: $T(n) = O(n^2)$
  - at most n calls to Partition (one for each item as pivot)
  - `Partition` requires $O(n)$ time

- This worst-case upper bound is tight:
  - If every time `Partition` is called *pivot* is greater (or smaller) than every other item, then its input is split into two lists, one of which has size 0.
    - This partitioning is very "unbalanced": let $c, d > 0$ be constants, where $T(0) = d$; then

$$T(n) = T(n-1) + T(0) + cn = \Theta(n^2).$$

△ A worst-case input is the sorted input!

Running time: average case analysis

Average case: what is an "average" input to sorting?

- Depends on the application.
- Inntuition why average-case analysis for uniformly distributed inputs to `Quicksort` is $O(n \log n)$ appears in your textbook.
- We will use randomness within the algorithm to provide `Quicksort` with a uniform at random input.

Today

Two views of randomness in computation

1. Deterministic algorithm, randomness over the inputs
   - On the same input, the algorithm always produces the same output using the same time.
     - So far, we have only encountered such algorithms.
   - The input is randomly generated according to some underlying distribution.
   - Average case analysis: analysis of the running time of the algorithm on an average input.

Two views of randomness in computation (cont'd)

(randomize inside the algorithm)

2. Randomized algorithm, worst-case (deterministic) input

- On the same input, the algorithm produces the same output but different executions may require different running times.
  - The latter depend on the random choices of the algorithm (e.g., coin flips, random numbers).
  - Random samples are assumed independent of each other.
- Worst-case input
- Expected running time analysis: analysis of the running time of the randomized algorithm on a worst-case input.

Randomized Algorithms are either one of the following:

Type 1: (Las Vegas)

- Deterministically correct; and
- Running-time is a random variable.

Type 2: (Monte Carlo)

- Probabilistically correct
- Deterministically fast (Running in polynomial time).

Remarks on randomness in computation

1. Deterministic algorithms are a special case of randomized algorithms.
2. Even when equally efficient deterministic algorithms exist, randomized algorithms may be simpler, require less memory of the past or be useful for symmetry-breaking.

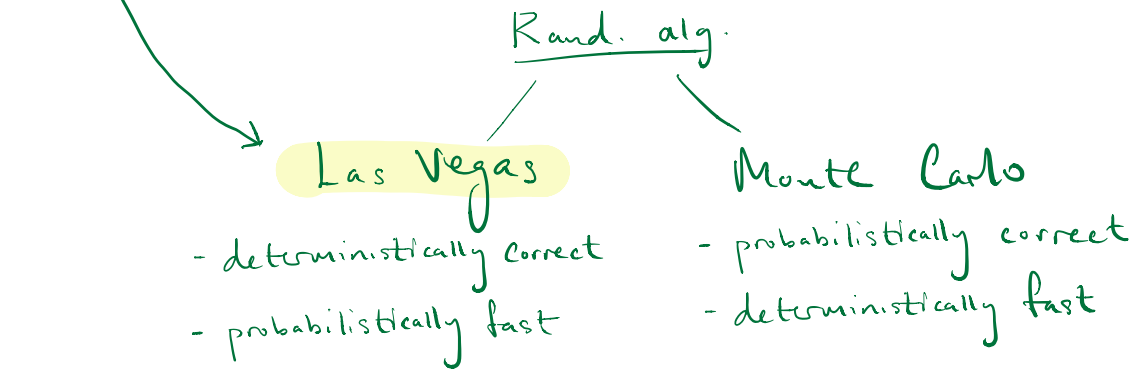
Randomized algorithms are either one of 2 types:

Type 1 (Las Vegas algorithms):
- Deterministically correct; and
- Running-time is a random variable.

Type 2 (Monte Carlo):
- Probabilistically correct; and
- Deterministically fast (Running in polynomial time).

- For Las Vegas algorithms, we know the output will be correct, but the running time will be fast (polynomial time) in expectation.

- Randomized QuickSort is a Las Vegas algorithm. (See next slide).



Randomized Quicksort

Can we use randomization so that Quicksort works with an "average" input even when it receives a worst-case input?

Do one of the following:

Do one or the other (not both)

1. Explicitly permute the input.
2. Use random sampling to choose *pivot*: instead of using $A[right]$ as *pivot*, select *pivot* randomly.

↑ Instead of using a fixed pivot item.

Idea 1 (intuition behind random sampling).

No matter how the input is organized, we won't often pick the largest or smallest item as pivot (unless we are really, really unlucky). Thus most often the partitioning will be "balanced".

Pseudocode for randomized Quicksort

```
Randomized-Quicksort(A, left, right)
    if |A| == 0 then return //A is empty
    end if
    split = Randomized-Partition(A, left, right)
    Randomized-Quicksort(A, left, split − 1)
    Randomized-Quicksort(A, split + 1, right)
```

```
Randomized-Partition(A, left, right)
    b = random(left, right)
    swap(A[b], A[right])
    return Partition(A, left, right)
```

the only modification to Partition(·)

The running time depends on the random choice b.

Subroutine random(i, j) returns a random number between i and j inclusive.

↳ O(1)

b = random(left, right) generates a random index.

Today

Discrete random variables

- To analyze the expected running time of a randomized algorithm we keep track of certain parameters and their expected size over the random choices of the algorithm.
- To this end, we use random variables.
- A discrete random variable X takes on a finite number of values, each with some probability. We're interested in its expectation

$$E[X] = \sum_j j \cdot \Pr[X = j].$$

Example 1: Bernoulli trial

Experiment 1: flip a biased coin which comes up

- *heads* with probability p
- *tails* with probability $1 - p$

Question: what is the expected number of *heads*?

Let $X = \begin{cases} 1, & \text{if coin flip comes up H} \\ 0, & \text{if coin flip comes up T} \end{cases}$

$$\mathbb{E}[X] = \sum_{j=0}^{1} j \Pr[X = j] = 0 + \Pr[X = 1]$$

$$= \Pr[X = 1] = p$$

The expected value of an indicator variable is the probability that the indicator variable takes on the value 1.

Example 1: Bernoulli trial

Experiment 1: flip a biased coin which comes up

- *heads* with probability p
- *tails* with probability $1 - p$

Question: what is the expected number of *heads*?

Let X be a random variable such that

$$X = \begin{cases} 1 & \text{, if coin flip comes } \textit{heads} \\ 0 & \text{, if coin flip comes } \textit{tails} \end{cases}$$

Example 1: Bernoulli trial

Experiment 1: flip a biased coin which comes up

- *heads* with probability p
- *tails* with probability $1 - p$

Question: what is the expected number of *heads*?

Let X be a random variable such that

$$X = \begin{cases} 1 & \text{, if coin flip comes } \textit{heads} \\ 0 & \text{, if coin flip comes } \textit{tails} \end{cases}$$

Then

$$\Pr[X = 1] = p$$
$$\Pr[X = 0] = 1 - p$$
$$E[X] = 1 \cdot \Pr[X = 1] + 0 \cdot \Pr[X = 0] = p$$

Indicator random variables

- Indicator random variable: a discrete random variable that only takes on values 0 and 1.
- Indicator random variables are used to denote occurrence (or not) of an event.

  Example: in the biased coin flip example, X is an indicator random variable that denotes the occurrence of *heads*.

Fact 1.

If X is an indicator random variable, then $E[X] = \Pr[X = 1]$.

Example 2: Bernoulli trials

Experiment 2: flip the biased coin n times

Question: what is the expected number of *heads*?

Let X be a r.v. that counts the # of Heads in n coin flips.

$$\mathbb{E}[X] = \sum_{j=0}^{n} j \Pr[X=j] = \sum_{j=0}^{n} j \cdot \binom{n}{j} p^j (1-p)^{n-j}$$

Instead of computing this complicated sum, we can instead use an indicator r.v. :

Let $X_i = \begin{cases} 1, & \text{if } i^{th} \text{ coin flip is H} \\ 0, & \text{o.w.} \end{cases}$

We want $E[X]$, where X is a r.v. that counts the # of Heads in n coin flips.

Let $X = \sum_{i=1}^{n} X_i$ Now X counts the # of heads.

$$E[X] = E\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} E[X_i]$$

$$= \sum_{i=1}^{n} Pr[X_i = 1] = np$$

Example 2: Bernoulli trials

Experiment 2: flip the biased coin n times

Question: what is the expected number of *heads*?

Answer 1: Let X be the random variable counting the number of times *heads* appears.

$$E[X] = \sum_{j=0}^{n} j \cdot \Pr[X = j].$$

$\Pr[X = j]$?

Example 2: Bernoulli trials

Experiment 2: flip the biased coin n times

Question: what is the expected number of *heads*?

Answer 1: Let X be the random variable counting the number of times *heads* appears.

$$E[X] = \sum_{j=0}^{n} j \cdot \Pr[X = j].$$

$\Pr[X = j]$?

X follows the binomial distribution $B(n, p)$, thus

$$\Pr[X = j] = \binom{n}{j} p^j (1-p)^{n-j}$$

Example 2: Bernoulli trials

A different way to think about X:

Answer 2: for $1 \leq i \leq n$, let X_i be an indicator random variable such that

$$X_i = \begin{cases} 1 & \text{, if } i\text{-th coin flip comes } \textit{heads} \\ 0 & \text{, if } i\text{-th coin flip comes } \textit{tails} \end{cases}$$

Example 2: Bernoulli trials

A different way to think about X:

Answer 2: for $1 \leq i \leq n$, let X_i be an indicator random variable such that

$$X_i = \begin{cases} 1 & \text{, if } i\text{-th coin flip comes } \textit{heads} \\ 0 & \text{, if } i\text{-th coin flip comes } \textit{tails} \end{cases}$$

Define the random variable

$$X = \sum_{i=1}^{n} X_i$$

We want $E[X]$. By Fact 1, $E[X_i] = p$, for all i.

Linearity of expectation

$$X = \sum_{i=1}^{n} X_i, \quad E[X_i] = p, \quad E[X] = ?$$

Linearity of expectation

$$X = \sum_{i=1}^{n} X_i, \quad E[X_i] = p, \quad E[X] = ?$$

Remark 1: X is a complicated random variable defined as the sum of simpler random variables whose expectation is known.

Linearity of expectation

$$X = \sum_{i=1}^{n} X_i, \quad E[X_i] = p, \quad E[X] = ?$$

Remark 1: X is a complicated random variable defined as the sum of simpler random variables whose expectation is known.

Proposition 1 (Linearity of expectation).

Let $X_1, \ldots, X_k$ be arbitrary random variables. Then

$$E[X_1 + X_2 + \ldots + X_k] = E[X_1] + E[X_2] + \ldots + E[X_k]$$

Linearity of expectation

$$X = \sum_{i=1}^{n} X_i, \quad E[X_i] = p, \quad E[X] = ?$$

Remark 1: X is a complicated random variable defined as the sum of simpler random variables whose expectation is known.

Proposition 1 (Linearity of expectation).

Let $X_1, \ldots, X_k$ be arbitrary random variables. Then

$$E[X_1 + X_2 + \ldots + X_k] = E[X_1] + E[X_2] + \ldots + E[X_k]$$

Remark 2: We made no assumptions on the random variables. For example, they do not need be independent.

Back to example 2: Bernoulli trials

Answer 2: for $1 \leq i \leq n$, let X_i be an indicator random variable such that

$$X_i = \begin{cases} 1 & \text{, if } i\text{-th coin flip comes } \textit{heads} \\ 0 & \text{, if } i\text{-th coin flip comes } \textit{tails} \end{cases}$$

Define the random variable

$$X = \sum_{i=1}^{n} X_i$$

By Fact 1, $E[X_i] = p$, for all i. By linearity of expectation,

$$E[X] = E\Big[\sum_{i=1}^{n} X_i\Big] = \sum_{i=1}^{n} E[X_i] = \sum_{i=1}^{n} p = np.$$

Today

Expected running time of randomized Quicksort

- Let $T(n)$ be the expected running time of `Randomized-Quicksort`.
- We want to bound $T(n)$.
- `Randomized-Quicksort` differs from `Quicksort` only in how they select their pivot elements.

$\Rightarrow$ We will analyze `Randomized-Quicksort` based on `Quicksort` and `Partition`.

Pseudocode for Partition

- n Partition calls in any execution of QS.
- n Partition calls in any execution of Randomized QS.

```
Partition(A, left, right)
pivot = A[right]                                   line 1
split = left − 1                                   line 2
for j = left to right − 1 do                       line 3
    if A[j] ≤ pivot then                           line 4
        swap(A[j], A[split + 1])                   line 5
        split = split + 1                          line 6
    end if
end for
swap(pivot, A[split + 1])                          line 7
return split + 1                                   line 8
```

O(1) O(right − left) O(1)

- Total time spent in ALL Partition calls outside the for-loop: $O(n)$
- Total time spent in ALL Partition calls inside the for-loop: $O\left(\sum_{i=1}^{n} X_i\right)$

Focus on 1 Partition Call first: $= O(X)$

Inside the for-loop: One comparison is made in each iteration. The for-loop iterates right-left times, (hence the O(right-left) run-time for the for-loop). Therefore, right-left is the same (or equal to) as the # of comparisons you are performing in one Partition call.

Let X_i = # comparisons performed in the i^{th} partition call.

Then we can express the time inside the for-loop in terms of the X_i.

Therefore, the time spent in the for-loop in Partition call i is: $O(X_i)$

(Since we know that every time we enter the for-loop we perform 1 comparison.)

Total time spent in ALL Partition calls inside the for-loop: $O\left(\sum_{i=1}^{n} X_i\right) = O(X)$, where we let $X = \sum_{i=1}^{n} X_i$ (i.e. X is the total # of comparisons in ALL Partition calls)

The running time of Randomized QuickSort is then expressed in terms of the # of comparisons performed.

The TOTAL Running time Randomized QuickSort is the time spent in ALL Partition calls outside the for-loop + the time spent in ALL Partition calls inside the for-loop:

$$\underbrace{O(n)}_{\text{outside for-loops}} + \underbrace{O(X)}_{\text{inside the for-loops.}} = O(n + X)$$

Running time of RandQS $= O(n + \overset{\text{r.v.}}{\uparrow} X)$

which is a random variable since X is a r.v.

Let $T(n)$ be the expected run-time of RandQS.

Then,

$$T(n) = \mathbb{E}\left[O(n) + O(X)\right]$$

$$= \mathbb{E}\left[O(n)\right] + \mathbb{E}\left[O(X)\right]$$

$$= O(n) + O(\mathbb{E}[X])$$

NOTE:
$O(X) = cX$
$\Rightarrow E[O(X)]$
$= O(E[X])$

- # of pairs of items (to compare): $\binom{n}{2}$ possible pairs of distinct items that are unordered.

- Fix two items to compare, say α_i and α_j.
How many times are they compared throughout the algorithm?

Answer: ≤ 1 (at most once).

Ex:

Now 1 and 7 will never be compared because they are placed in 2 separate partitions

Then, $X \leq \binom{n}{2} = O(n^2)$

Thus, in the worst case run-time of RandQS is $O(n + n^2) = O(n^2)$, but we know this.

Analyze the expectation of X instead

↓

Assume : all input items are distinct.

Relabel input items to $z_1, z_2, \ldots, z_n$ s.t. $z_1 < z_2 < \cdots < z_n$.

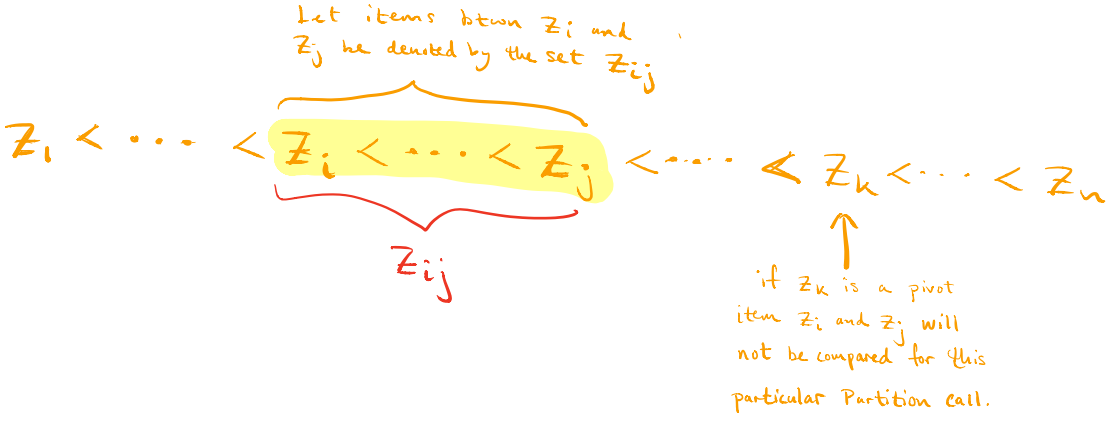
Let $X_{ij} = \begin{cases} 1, & \text{if } z_i \text{ and } z_j \text{ are ever compared} \\ 0, & \text{otherwise} \end{cases}$

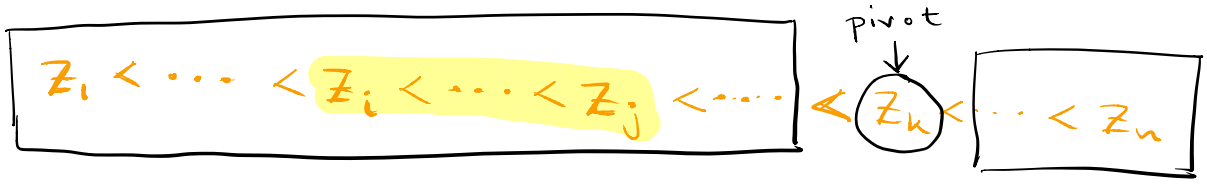
Then, $X = \sum_{1 \le i < j \le n} X_{ij}$;

$$\mathbb{E}[X] = \mathbb{E}\left[\sum_{1 \le i < j \le n} X_{ij}\right] = \sum_{1 \le i < j \le n} \mathbb{E}[X_{ij}]$$

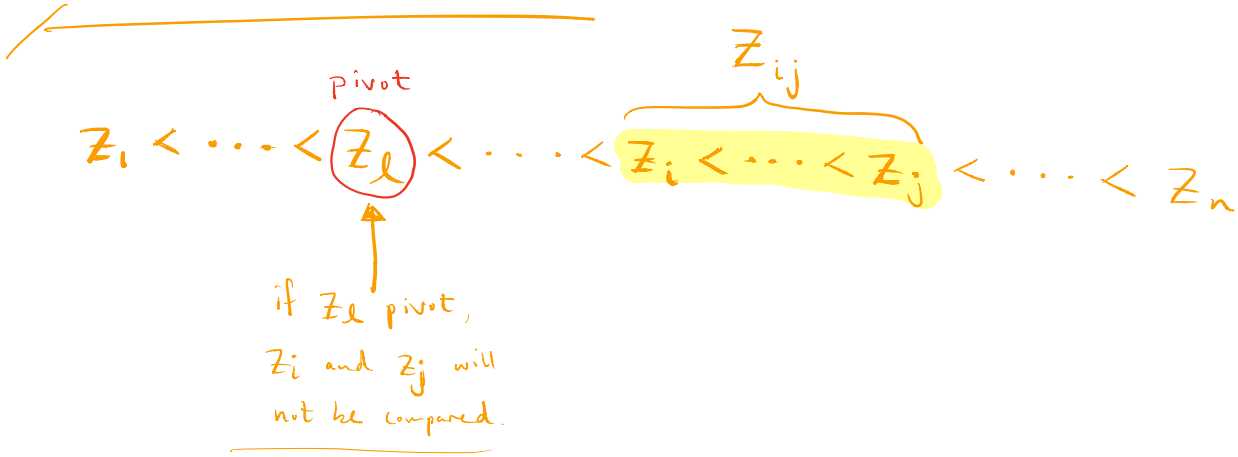
$$= \sum_{1 \le i < j \le n} \Pr[X_{ij} = 1]$$

To determine $\Pr[X_{ij} = 1]$ for fixed z_i and z_j consider :





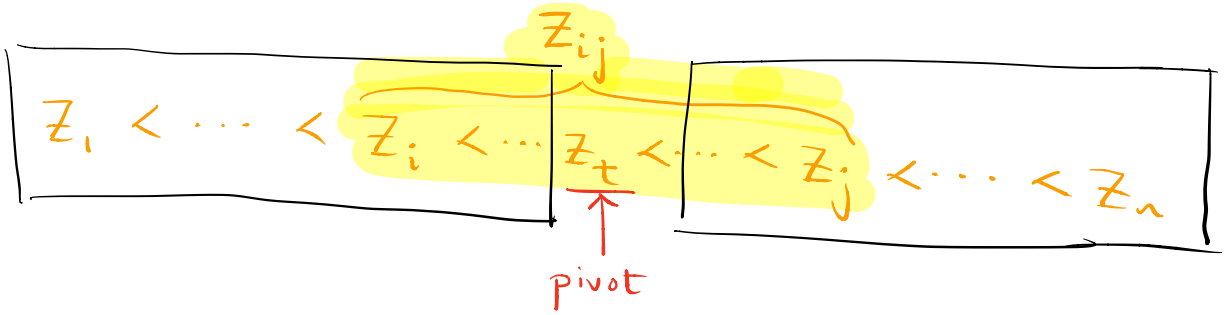
- Still possible for z_i and z_j to be compared in a future Partition call (but are not compared in this particular call).



In the first Partition call that picks its pivot item from the set Z_{ij}, we can determine whether z_i and z_j will be compared or not, and we can compute the exact probability w/ which they will be compared.

- If z_i or z_j is picked as the pivot then they will definitely be compared.
- If, however, z_t is the pivot where $z_t \neq z_i$ and $z_t \neq z_j$ then z_i and z_j will not be compared.

That is the probability of z_i and z_j being compared when $z_t \in Z_{ij}$ is the pivot item where $z_t \neq z_i$ and $z_t \neq z_j$ is zero.



Therefore, z_i and z_j never compared when z_t is the pivot since z_i and z_j will be partitioned into a different subproblem.

The first Partition call that picks its pivot from the set Z_{ij} determines the value of X_{ij}.

Therefore we can compute the probability as follows:

$$\Pr\left[X_{ij} = 1 \;\middle|\; \text{pivot} \in Z_{ij} \text{ in this Partition call}\right]$$

$$= \Pr\left[Z_i \text{ or } Z_j \text{ is picked as pivot} \;\middle|\; \text{pivot} \in Z_{ij} \text{ in this Partition call}\right]$$

$$= \Pr\left[Z_i \text{ is picked as pivot} \;\middle|\; \text{pivot} \in Z_{ij} \text{ in this Partition call}\right] +$$

$$+ \Pr\left[Z_j \text{ is picked as pivot} \;\middle|\; \text{pivot} \in Z_{ij} \text{ in this Partition call}\right]$$

$$= \frac{1}{|Z_{ij}|} + \frac{1}{|Z_{ij}|} = \frac{2}{|Z_{ij}|} = \frac{2}{j-i+1}$$

NOTE:

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

This is going to be the same for every Partition call.

$$\Pr\left[X_{ij} = 1 \;\middle|\; \text{pivot} \in Z_{ij} \text{ in this Partition call}\right] = \frac{2}{j-i+1}$$

for every Partition call.

Therefore,

$$\Pr[X_{ij} = 1] = \sum_{\text{Partition calls}} \Pr\left[X_{ij} = 1 \;\middle|\; \text{pivot} \in Z_{ij} \text{ in this Partition call}\right] \cdot$$

$$\cdot \Pr\left[\text{pivot} \in Z_{ij} \text{ in this Partition call}\right]$$

$$= \frac{2}{j-i+1} \cdot \underbrace{\sum_{\text{Partition Calls}} \Pr\left[\begin{matrix}\text{pivot} \in Z_{ij} \text{ in} \\ \text{this Partition call}\end{matrix}\right]}$$

$$= 1$$

Since an item from Z_{ij} will for sure be picked @ some point and we are summing over ALL Partition Calls.

$$= \frac{2}{j-i+1}$$

Now,

$$\mathbb{E}[X] = \sum_{1 \le i < j \le n} \frac{2}{j-i+1}$$

$$= \sum_{i=1}^{n-1} \sum_{j=i}^{n} \frac{2}{j-i+1}$$

Set $k = j-i+1$

For $j=i \Rightarrow k=1$

For $j=n \Rightarrow k=n-i+1$

$$= 2 \sum_{i=1}^{n-1} \sum_{k=1}^{n-i+1} \frac{1}{k}$$

UNION BOUND:

$\Pr\{\mathcal{E}_1 \cup \mathcal{E}_2\} \le \Pr\{\mathcal{E}_1\} + \Pr\{\mathcal{E}_2\}$

If $\mathcal{E}_1 \cap \mathcal{E}_2 = \emptyset$ (MUTUALLY EXCLUSIVE)

$\Pr\{\mathcal{E}_1 \cup \mathcal{E}_2\} = \Pr\{\mathcal{E}_1\} + \Pr\{\mathcal{E}_2\}$

$$\leq 2 \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{1}{k}$$

$$\leq 2nH_n$$

$$\simeq 2n(\ln(n) + \gamma)$$

$$= O(n \log(n))$$

This is a tight bound.

Therefore, the expected running time of Randomized QuickSort is $\Theta(n \log(n))$.

- Other ways to analyze the expected running time of Randomized QuickSort.

(1.) Derive a recurrence for the expected run time, and analyze the running time by analyze the run time of that recurrence.

(2.) View what we did as a game.

We show both below:

(1.) To evaluate the expected running time using a recurrence, consider the recurrence for the original QuickSort algorithm:

$$T(n) = cn + T(\text{split} - 1) + T(n - \text{split})$$

In the Randomized QuickSort algorithm, split is a random variable. Therefore, $T(\text{split} - 1)$ is a r.v. and $T(n - \text{split})$ is a r.v.

And Split is a uniform r.v.. Then, ↘

$$T(n) = cn + \frac{1}{n}\left(T(n-1) + T(0)\right)$$
$$+ \frac{1}{n}\left(T(n-2) + T(1)\right)$$
$$+ \frac{1}{n}\left(T(n-3) + T(2)\right)$$
$$\vdots \qquad\qquad \vdots$$
$$+ \frac{1}{n}\left(T(0) + T(n-1)\right)$$

Then evaluate

(2) To analyze using a game, consider the following game:

- Game of darts

- Game ends if you hit a number in z_{ij}
  - If you hit z_i or z_j, you WIN
  - If you hit some z_ℓ s.t. $z_i < z_\ell < z_j$, you LOSE.

U ——

- If you hit a number **NOT** in Z_{ij}, remove Z_k and get to throw another dart.

$$\Pr[\text{WIN}] = \sum_{\text{rounds}} \Pr[\text{Hit } z_i \text{ or } z_j \mid \text{game terminates in this round}] \cdot \Pr[\text{Game terminates in this round}]$$

$$= \sum_{\text{rounds}} \frac{2}{j-i+1} \cdot \Pr[\text{Game terminates in this round}]$$

$$= \frac{2}{j-i+1} \sum_{\text{rounds}} \Pr[\text{Game terminates in this round}]$$

$$= \frac{2}{j-i+1} \cdot 1 = \frac{2}{j-i+1}$$

Few observations

1. *How many times is* `Partition` *called?*

Few observations

1. *How many times is* `Partition` *called?*
At most n.

2. Further, each `Partition` call spends some work
   1. outside the for loop

   2. inside the for loop

Few observations

1. *How many times is* `Partition` *called?*
At most n.

2. Further, each `Partition` call spends some work
   1. outside the for loop
      - every `Partition` spends constant work outside the for loop
      - at most n calls to `Partition`

      $\Rightarrow$ total work outside the for loop in all calls to `Partition` is $O(n)$
   2. inside the for loop

Few observations

1. *How many times is* `Partition` *called?*
At most n.

2. Further, each `Partition` call spends some work
   1. outside the for loop
      - every `Partition` spends constant work outside the for loop
      - at most n calls to `Partition`

      $\Rightarrow$ total work outside the for loop in all calls to `Partition` is $O(n)$
   2. inside the for loop
      - let X be the total number of comparisons performed at line 4 in all calls to `Partition`
      - each comparison may require some further constant work (lines 5 and 6)

      $\Rightarrow$ total work inside the for loop in all calls to `Partition` is $O(X)$

Towards a bound for $T(n)$

$X =$ the total number of comparisons in **all** `Partition` calls.

The running time of `Randomized-Quicksort` is

$$O(n + X).$$

Since X is a random variable, we need $E[X]$ to bound $T(n)$.

Towards a bound for $T(n)$

$X =$ the total number of comparisons in **all** `Partition` calls.

The running time of `Randomized-Quicksort` is

$$O(n + X).$$

Since X is a random variable, we need $E[X]$ to bound $T(n)$.

Fact 2.

Fix any two input items. During the execution of the algorithm, they may be compared at most once.

Towards a bound for $T(n)$

X = the total number of comparisons in **all** `Partition` calls.

The running time of `Randomized-Quicksort` is

$$O(n + X).$$

Since X is a random variable, we need $E[X]$ to bound $T(n)$.

Fact 2.

Fix any two input items. During the execution of the algorithm, they may be compared at most once.

Proof.

Comparisons are only performed with the *pivot* of each `Partition` call. After `Partiton` returns, *pivot* is in its final location in the output and will not be part of the input to any future recursive call. □

Simplifying the analysis

- There are n numbers in the input, hence $\binom{n}{2} = \frac{n(n-1)}{2}$ distinct (unordered) pairs of input numbers.
- From Fact 1, the algorithm will perform at most $\binom{n}{2}$ comparisons.
- *What is the expected number of comparisons?*

Simplifying the analysis

- There are n numbers in the input, hence $\binom{n}{2} = \frac{n(n-1)}{2}$ distinct (unordered) pairs of input numbers.
- From Fact 1, the algorithm will perform at most $\binom{n}{2}$ comparisons.
- *What is the expected number of comparisons?*

To simplify the analysis

- relabel the input as $z_1, z_2, \ldots, z_n$, where z_i is the i-th smallest number.
- **assume** that all input numbers are distinct; thus $z_i < z_j$, for $i < j$.

Writing X as the sum of indicator random variables

Let X_{ij} be an indicator random variable such that

$$X_{ij} = \begin{cases} 1, & \text{if } z_i \text{ and } z_j \text{ are ever compared} \\ 0, & \text{otherwise} \end{cases}$$

Writing X as the sum of indicator random variables

Let X_{ij} be an indicator random variable such that

$$X_{ij} = \begin{cases} 1, & \text{if } z_i \text{ and } z_j \text{ are ever compared} \\ 0, & \text{otherwise} \end{cases}$$

The total number of comparisons is given by $X = \sum_{1 \le i < j \le n} X_{ij}$.

Writing X as the sum of indicator random variables

Let X_{ij} be an indicator random variable such that

$$X_{ij} = \begin{cases} 1, & \text{if } z_i \text{ and } z_j \text{ are ever compared} \\ 0, & \text{otherwise} \end{cases}$$

The total number of comparisons is given by $X = \sum_{1 \le i < j \le n} X_{ij}$.

$E[X] = ?$

Writing X as the sum of indicator random variables

Let X_{ij} be an indicator random variable such that

$$X_{ij} = \begin{cases} 1, & \text{if } z_i \text{ and } z_j \text{ are ever compared} \\ 0, & \text{otherwise} \end{cases}$$

The total number of comparisons is given by $X = \sum_{1 \le i < j \le n} X_{ij}$.

By linearity of expectation

$$E[X] = E\left[\sum_{1 \le i < j \le n} X_{ij}\right] = \sum_{1 \le i < j \le n} E[X_{ij}] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr[X_{ij} = 1]$$

Writing X as the sum of indicator random variables

Let X_{ij} be an indicator random variable such that

$$X_{ij} = \begin{cases} 1, & \text{if } z_i \text{ and } z_j \text{ are ever compared} \\ 0, & \text{otherwise} \end{cases}$$

The total number of comparisons is given by $X = \sum_{1 \le i < j \le n} X_{ij}$.

By linearity of expectation

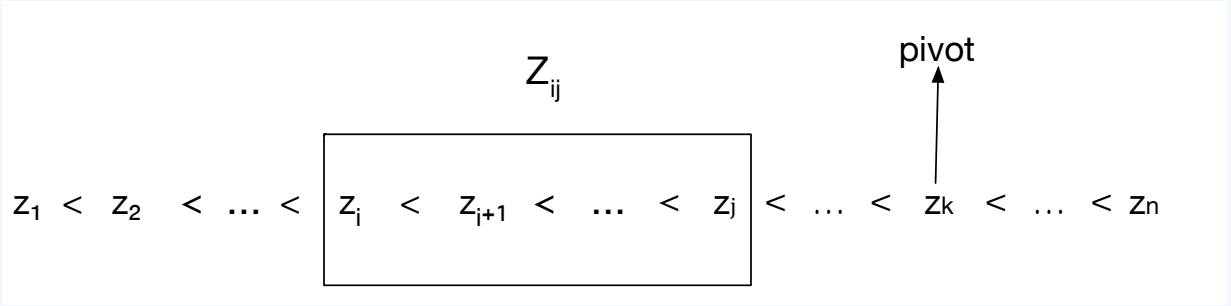
$$E[X] = E\left[\sum_{1 \le i < j \le n} X_{ij}\right] = \sum_{1 \le i < j \le n} E[X_{ij}] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr[X_{ij} = 1]$$

Goal: compute $\Pr[X_{ij} = 1]$, that is, the probability that two *fixed* items z_i and z_j are ever compared.

Fix two items z_i and z_j. When are they compared?

Notation: let $Z_{ij} = \{z_i, z_{i+1}, \ldots, z_j\}$

Consider the initial call `Partition`$(A, 1, n)$. Assume it picks z_k **outside** Z_{ij} as *pivot* (see figure below).



1. z_i and z_j are **not** compared in this call *(why?)*.
2. All items in Z_{ij} will be greater (or smaller) than z_k, so they will **all be input to the same subproblem** after `Partition`$(A, 1, n)$ returns.

In the first `Partition` with $pivot \in Z_{ij} = \{z_i, \ldots, z_j\}$

The first `Partition` call that picks its $pivot$ from Z_{ij} determines if z_i, z_j are ever compared. Three possibilities:

1. $pivot = z_i$

2. $pivot = z_j$

3. $pivot = z_\ell$, for some $i < \ell < j$

In the first `Partition` with $pivot \in Z_{ij} = \{z_i, \ldots, z_j\}$

The first `Partition` call that picks its *pivot* from Z_{ij} determines if z_i, z_j are ever compared. Three possibilities:

1. $pivot = z_i$

   z_i is compared with every element in $Z_{ij} - \{z_i\}$, thus with z_j too. z_i is placed in its final location in the output and will not appear in any future calls to `Partition`.

2. $pivot = z_j$

   z_j is compared with every element in $Z_{ij} - \{z_j\}$, thus with z_i too. z_j is placed in its final location in the output and will not appear in any future recursive calls.

3. $pivot = z_\ell$, for some $i < \ell < j$

   z_i and z_j are never compared *(why?)*

So z_i and z_j are compared when ...

... either of them is chosen as *pivot* in that first Partition call that chooses its *pivot* element from Z_{ij}.

Now we can compute $\Pr[X_{ij} = 1]$:

$$\begin{aligned}\Pr[X_{ij} = 1] \quad = \quad & \Pr[z_i \text{ is chosen as } \textit{pivot} \text{ by the first } \texttt{Partition} \\ & \quad \text{that picks its } \textit{pivot} \text{ from } Z_{ij}, \quad \textbf{or} \\ & \quad z_j \text{ is chosen as } \textit{pivot} \text{ by the first } \texttt{Partition} \\ & \quad \text{that picks its } \textit{pivot} \text{ from } Z_{ij}] \qquad (1)\end{aligned}$$

The union bound

Suppose we are given a set of events $\varepsilon_1, \varepsilon_2, \ldots, \varepsilon_n$, and we are interested in the probability that **any** of them happens.

Union bound: Given events $\varepsilon_1, \varepsilon_2, \ldots, \varepsilon_n$, we have

$$\Pr\left[\bigcup_{i=1}^{n} \varepsilon_i\right] \leq \sum_{i=1}^{n} \Pr[\varepsilon_i].$$

Union bound for mutually exclusive events: Suppose that $\varepsilon_i \cap \varepsilon_j = \emptyset$ for each pair of events. Then

$$\Pr\left[\bigcup_{i=1}^{n} \varepsilon_i\right] = \sum_{i=1}^{n} \Pr[\varepsilon_i].$$

Computing the probability that z_i and z_j are compared

Since the two events in equation (1) are mutually exclusive, we obtain

$$\begin{aligned}
\Pr[X_{ij} = 1] &= \Pr[z_i \text{ is chosen as } \textit{pivot} \text{ by the first } \texttt{Partition} \\
&\qquad \text{call that picks its } \textit{pivot} \text{ from } Z_{ij}] \\
&+ \Pr[z_j \text{ is chosen as } \textit{pivot} \text{ by the first } \texttt{Partition} \\
&\qquad \text{call that picks its } \textit{pivot} \text{ from } Z_{ij}] \\
&= \frac{1}{j-i+1} + \frac{1}{j-i+1} = \frac{2}{j-i+1}, \qquad (2)
\end{aligned}$$

since the set Z_{ij} contains $j - i + 1$ elements.

From $\Pr[X_{ij} = 1]$ to $E[X]$

$$\begin{aligned} E[X] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr[X_{ij} = 1] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \\ &= 2 \sum_{i=1}^{n-1} \sum_{\ell=2}^{n-i+1} \frac{1}{\ell} \end{aligned} \tag{3}$$

Note that $\sum_{\ell=1}^{k} \frac{1}{\ell} = H_k$ is the k-th harmonic number, such that

$$\ln k \leq H_k \leq \ln k + 1 \tag{4}$$

Hence $\sum_{\ell=2}^{n-i+1} \frac{1}{\ell} \leq \ln (n-i+1)$. Substituting in (3), we get

$$E[X] \leq 2 \sum_{i=1}^{n-1} \ln (n-i+1) \leq 2 \sum_{i=1}^{n-1} \ln n = O(n \ln n)$$

From $E[X]$ to $T(n)$

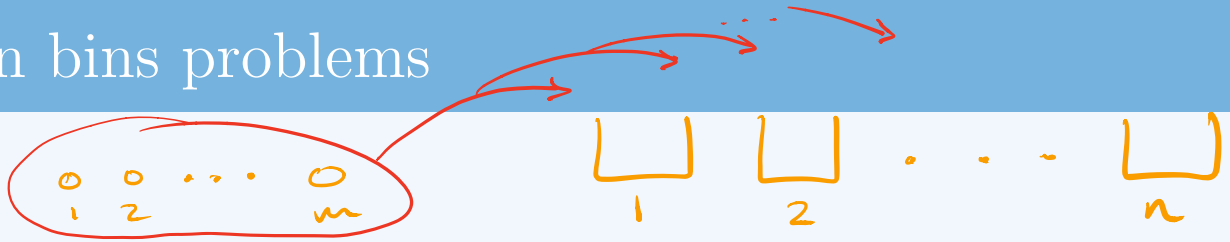
- Equations (3), (4) also yield a lower bound of $\Omega(n \ln n)$ for $E[X]$ *(show this!)*.

- Hence $E[X] = \Theta(n \ln n)$. Then the expected running time of `Randomized-Quicksort` is

$$T(n) = \Theta(n \ln n)$$

Today

Balls in bins problems



Occupancy problems: find the distribution of balls into bins when m balls are thrown independently and uniformly at random into n bins.

- Applications: analysis of randomized algorithms and data structures (e.g., hash table)

Q1: How many balls can we throw before it is more likely than not that some bin contains at least two balls?

In symbols: *find k such that*

$$\Pr[\exists \text{ bin with } \geq 2 \text{ balls after } k \text{ balls thrown}] > 1/2$$

In symbols: *find k such that*

$$\Pr[\exists \text{ bin with } \geq 2 \text{ balls after } k \text{ balls thrown}] > 1/2$$

Equivalent, find k s.t.

$\Pr[\text{every bin has } \leq 1 \text{ ball after } k \text{ balls thrown}] \leq \frac{1}{2}$

Let $\mathcal{E}_i \equiv$ the event where every bin has ≤ 1 balls after i balls are thrown.

$\Pr[\mathcal{E}_1] = 1$. if throw 1 ball then every bin has at most one ball for sure.

$\Pr[\mathcal{E}_2] = 1 - \frac{1}{n}$

$\Pr[\mathcal{E}_3 \mid \mathcal{E}_2] = 1 - \frac{2}{n}$

$\vdots$

$\Pr[\mathcal{E}_k \mid \mathcal{E}_{k-1}] = 1 - \frac{k-1}{n}$

k^{th} ball must land in a new bin

NOTE: $\forall\, x \in \mathbb{R},\ 1 + x \leq e^x$

$$\Pr[\mathcal{E}_k] = \Pr[\mathcal{E}_k \mid \mathcal{E}_{k-1}] \cdot \Pr[\mathcal{E}_{k-1}]$$

$$= \Pr[\mathcal{E}_k \mid \mathcal{E}_{k-1}] \cdot \Pr[\mathcal{E}_{k-1} \mid \mathcal{E}_{k-2}] \cdot \Pr[\mathcal{E}_{k-2}]$$

$$\vdots$$

$$= \Pr[\mathcal{E}_k \mid \mathcal{E}_{k-1}] \cdot \Pr[\mathcal{E}_{k-1} \mid \mathcal{E}_{k-2}] \cdots \Pr[\mathcal{E}_2]$$

$$= \left(1 - \frac{k-1}{n}\right)\left(1 - \frac{k-2}{n}\right) \cdots \left(1 - \frac{1}{n}\right)$$

$$= \prod_{i=1}^{k-1} \left(1 - \frac{i}{n}\right)$$

Using the fact $\forall x \in \mathbb{R},\ 1 + x \le e^x$

$$\le \prod_{i=1}^{k-1} e^{-\frac{i}{n}} = e^{-\sum_{i=1}^{k-1} \frac{i}{n}}$$

$$= e^{-\frac{1}{n}\sum_{i=1}^{k-1} i} = e^{-\frac{1}{n} \cdot \frac{k(k-1)}{2}}$$

WE WANT: $\Pr[\mathcal{E}_k] < \frac{1}{2}$

$\Rightarrow$ Want to show: $e^{-\frac{k(k-1)}{2n}} < \frac{1}{2}$ ↘

$$\Rightarrow \quad 2 < e^{\frac{k(k-1)}{2n}}$$

$$\Rightarrow \quad \ln 2 < \frac{k(k-1)}{2n}$$

$$\Rightarrow \quad 2n \ln 2 < k(k-1) < k^2$$

$$\Rightarrow \quad k = \underline{O}(\sqrt{n}) = \Omega(\sqrt{n})$$

This says that as long as you throw at least $\Omega(\sqrt{n})$ balls, then it is more likely than not that some bin has at least 2 balls.

Rewatch lectures:

3/11, 3/16, 3/18, and 3/23

Easier to analyze the complement of this event

Easier to think about the probability of the complementary event.

Q1 (rephrased): Find k such that

$$\Pr[\textbf{every} \text{ bin has} \leq 1 \text{ ball after } k \text{ balls thrown}] \leq 1/2$$

Analysis: one ball at a time

- The 1st ball falls into some bin.
- The 2nd ball falls into a new bin w. prob. $1 - \frac{1}{n}$.
- The 3rd ball falls into a new bin given that the first two balls fell into different bins w. prob. $1 - \frac{2}{n}$.
- The m-th ball falls into a new bin given that the first $k-1$ balls fell into different bins w. prob. $1 - \frac{k-1}{n}$.

By the chain rule of conditional probability, the probability that the k-th ball falls into a new bin is given by

$$\prod_{i=1}^{k-1} \left(1 - \frac{i}{n}\right) \tag{5}$$

Application: the birthday paradox

Use $1 + x \leq e^x$ for all real x to upper bound (5)

$$\prod_{i=1}^{k-1} e^{-i/n} = e^{-\sum_{i=1}^{k-1} i/n} = e^{-\frac{k(k-1)}{(2 \cdot n)}} \approx e^{-\frac{k^2}{2n}} \tag{6}$$

Requiring $e^{-\frac{k^2}{2n}} < 1/2$ yields $k > \sqrt{n \cdot 2 \ln 2} = \Omega(\sqrt{n})$.

- **Application:** birthday paradox
  Assumption: For $n = 365$, each person has an independent and uniform at random birthday from among the 365 days of the year.
  Once 23 people are in a room, it is more likely than not that two of them share a birthday.

More balls-in-bins questions

- *Q2: What is the expected load of a bin after m balls are thrown?* $\frac{m}{n}$
- *Q3: What is the expected #empty bins after m balls are thrown?*
- *Q4: What is the load of the fullest bin with high probability?*
- *Q5: What is the expected number of balls until* **every** *bin has at least one ball (Coupon Collector's Problem)?*

- *Q2: What is the expected load of a bin after m balls are thrown?*

Fix a bin.

Let X = # of balls in the fixed bin.

We want $E[X]$.

Let $X_i = \begin{cases} 1, & \text{if } i^{th} \text{ ball falls in bin.} \\ 0, & \text{o.w.} \end{cases}$

$$X = \sum_{i=1}^{m} X_i$$

$$E[X] = E\left[\sum_{i=1}^{m} X_i\right] = \sum_{i=1}^{m} E[X_i]$$

$$= \sum_{i=1}^{m} Pr[X_i = 1] = \sum_{i=1}^{m} \frac{1}{n} = \frac{m}{n}$$

Expected load of a bin

Suppose that m balls are thrown independently and uniformly at random into n bins. Fix a bin.

- Let X_i be an indicator r.v. such that $X_i = 1$ if and only if ball i falls in the fixed bin. Then

$$E[X_i] = \Pr[X_i = 1] = \frac{1}{n}.$$

The total #balls in the bin is given by $X = \sum_{i=1}^{m} X_i$. By linearity of expectation,

$$E[X] = \sum_{i=1}^{m} E[X_i] = m/n.$$

Since bins are symmetric, the expected load of any bin is m/n.

▸ *Q3: What is the expected #empty bins after m balls are thrown?*

Let Y = # of empty bins after m balls thrown.

$$Y_j = \begin{cases} 1, & \text{if bin } j \text{ is empty} \\ 0, & \text{o.w.} \end{cases} \quad \Rightarrow \quad Y = \sum_{j=1}^{n} Y_j$$

$$E[Y] = E\left[\sum_{j=1}^{n} Y_j\right] = \sum_{j=1}^{n} E[Y_j] =$$

$$= \sum_{j=1}^{n} Pr[Y_j = 1] = \sum_{j=1}^{n} \prod_{i=1}^{m} Pr[\{\text{ball } i \text{ does not land in bin } j\}]$$

NOTE: $Pr[Y_j = 1] = Pr[\{\text{ball 1 does not land in bin } j\}$
$\cap \{\text{ball 2 does not land in bin } j\}$
$\cap \{\text{ball 3 does not land in bin } j\}$
$\vdots$
$\cap \{\text{ball } m \text{ does not land in bin } j\}]$

$$= Pr\left[\bigcap_{i=1}^{m} \{\text{ball } i \text{ does not land in bin } j\}\right]$$

$$= \prod_{i=1}^{m} Pr[\{\text{ball } i \text{ does not land in bin } j\}]$$

$$= \left(1-\frac{1}{n}\right)\cdot\left(1-\frac{1}{n}\right)\cdots\left(1-\frac{1}{n}\right)$$

$$= \prod_{i=1}^{m}\left(1-\frac{1}{n}\right) = \left(1-\frac{1}{n}\right)^m \leq e^{-\frac{m}{n}}$$

Now:

$$\sum_{j=1}^{n} \Pr[Y_j = 1] = \sum_{j=1}^{n} \prod_{i=1}^{m} \Pr\left[\left\{\text{ball } i \text{ does not land in bin } j\right\}\right]$$

$$= \sum_{j=1}^{n}\left(1-\frac{1}{n}\right)^m \leq \sum_{j=1}^{n}\left(e^{-\frac{1}{n}}\right)^m = ne^{-\frac{m}{n}}$$

$$\Rightarrow \mathbb{E}[Y] \leq ne^{-\frac{m}{n}}$$

As n and m grow large, this approximation is pretty tight. In fact, if $m=n$ we get that approximately $\frac{n}{e}$ bins are empty (more than a third of the bins are empty). Thats $\frac{n}{e} > \frac{n}{3}$, so

> a third of the bins are empty.

Expected # empty bins

Suppose that m balls are thrown independently and uniformly at random into n bins. Fix a bin j.

- Let Y_j be an indicator r.v. such that $Y_j = 1$ if and only if bin j is empty.
- $\Pr[\text{ball } i \text{ does not fall in bin } j] = 1 - 1/n$
- $\Pr[\text{for all } i, \text{ ball } i \text{ does not fall in bin } j] = (1 - 1/n)^m$
- Hence $\Pr[Y_j = 1] = (1 - 1/n)^m$.

The number of empty bins is given by the random variable $Y = \sum_{j=1}^{n} Y_j$. By linearity of expectation

$$E[Y] = \sum_{j=1}^{n} E[Y_j] = n\left(1 - \frac{1}{n}\right)^m \approx ne^{-m/n}$$

Maximum load with high probability (case $m = n$)

Proposition 2.

When throwing n balls into n bins uniformly and independently at random, the maximum load in any bin is $\Theta(\ln n / \ln \ln n)$ with probability close to 1 as n grows large.

Two-sentence sketch of the proof.

1. Upper bound the probability that **any** bin contains more than k balls by a union bound: $\sum_{j=1}^{n} \sum_{\ell=k}^{n} \binom{n}{\ell} \left(\frac{1}{n}\right)^{\ell} \left(1 - \frac{1}{n}\right)^{n-\ell}$.
2. Compute the smallest possible k^* such that the probability above is less than $1/n$; the latter becomes negligible as n grows large.

□

- *Q5: What is the expected number of balls until* **every** *bin has at least one ball (Coupon Collector's Problem)?*

Goal: to have at least 1 ball in every bin.

Question: How many balls should be thrown in the bins in expectation until every bin has at least 1 ball?

* Success ≡ ball that lands in a new bin

* epoch ≡ sequence of balls that ends with a success.

* epoch j ≡ sequence of balls that starts after the $(j-1)^{st}$ success and ends with the j^{th} success.

* X_j ≡ # of balls in epoch j

Let X = # of balls thrown until every bin has ≥ 1 ball.

Then $X = \sum_{j=1}^{n} X_j$

We want $E[X]$.

$$E[X] = E\left[\sum_{j=1}^{n} X_j\right] = \sum_{j=1}^{n} E[X_j]$$

$X_1 = 1$ (deterministic)

$X_2 = ?$

$$\Pr[X_2 = k] = \underbrace{\left(\frac{1}{n}\right)^{k-1}}_{\text{Prob. that you keep hitting the 1st bin in the first } k-1 \text{ trials.}} \cdot \overbrace{\left(1 - \frac{1}{n}\right)}^{\text{success probability}}$$

$$\Pr[X_3 = k] = \left(\frac{2}{n}\right)^{k-1} \cdot \left(1 - \frac{2}{n}\right)$$

$$\vdots$$

$$\Pr[X_j = k] = \left(\frac{j-1}{n}\right)^{k-1} \cdot \left(1 - \frac{j-1}{n}\right)$$

The X_i are geometrically distributed random variables.

If $X \sim \text{Geometric}(k, p)$, then $E[X] = \frac{1}{p}$ (success probability)

and $\Pr[X=k] = (1-p)^{k-1} \cdot p$ ← success prob.

$$E[X] = \sum_{j=1}^{n} E[X_j] = \sum_{j=} \frac{1}{\Pr[\text{Success}]}$$

$$= \sum_{j=1}^{n} \frac{1}{\left(1 - \frac{j-1}{n}\right)} = \sum_{j=1}^{n} \frac{1}{\left(\frac{n-j+1}{n}\right)}$$

$$= \sum_{j=1}^{n} \frac{n}{n-j+1}$$

$1 - \frac{j-1}{n} = \frac{n-j+1}{n}$

Let $k = n-j+1$
For $j=1$, $k=n$
For $j=n$, $k=1$

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

$$= n \cdot H_n \quad (H_n \text{ is the } n^{\text{th}} \text{ harmonic \#})$$

$$\approx n \cdot (\ln(n) + \gamma)$$

γ: a constant

This means you need approx. $n\ln(n)$ balls in the system before every bin has at least 1 ball.

Expected #balls until no empty bins

Suppose that we throw balls independently and uniformly at random into n bins, one at a time (the first ball falls at time $t = 1$).

- We call a throw a **success** if it lands in an empty bin.
- We call the sequence of balls starting after the $(j-1)$-st success and ending with the j-th success, the j-th **epoch**.
- To understand when the process terminates, we need analyze the duration of each epoch.
- To this end, let η_j be the #balls thrown in epoch j.
- Clearly the first ball is a **success**, hence $\eta_1 = 1$.
- Let η_2 be the #balls thrown in epoch 2.

$$\forall t \in \text{epoch } 2, \Pr[\text{ball } t \text{ in epoch } 2 \text{ is a } \textbf{success}] = \frac{n-1}{n}$$

- Similarly, let η_j be the #balls thrown in epoch j.

$$\forall t \in \text{epoch } j, \Pr[\text{ball } t \text{ in epoch } j \text{ is a } \textbf{success}] = \frac{n-j+1}{n}$$

At the end of the n-th epoch, each of the n bins has at least one ball.

Expected #balls until no empty bins (cont'd)

Let $\eta = \sum_{j=1}^{n} \eta_j$. We want

$$E[\eta] = E\left[\sum_{j=1}^{n} \eta_j\right] = \sum_{j=1}^{n} E[\eta_j]$$

- Each epoch is geometrically distributed with success probability $p_j = \frac{n-j+1}{n}$.
- Recall that the expectation of a geometrically distributed variable with success probability p is given by $1/p$.
- Thus $E[\eta_j] = \frac{1}{p_j} = \frac{n}{n-j+1}$.

Then

$$E[\eta] = \sum_{j=1}^{n} \frac{n}{n-j+1} = n \sum_{j=1}^{n} \frac{1}{j} = n(\ln n + O(1))$$

Probability review

- A sample space Ω consists of the possible outcomes of an experiment.
- Each point x in the sample space has an associated probability mass $p(x) \geq 0$, such that $\sum_{x \in \Omega} p(x) = 1$.
- Example experiment: flip a fair coin;
  $\Omega = \{\mathit{heads}, \mathit{tails}\}$; $\Pr[\mathit{heads}] = \Pr[\mathit{tails}] = 1/2$.
- We define an event $\mathcal{E}$ to be any subset of Ω, that is, a collection of points in the sample space.
- We define the probability of the event to be the sum of the probability masses of all the points in $\mathcal{E}$. That is,

$$\Pr[\mathcal{E}] = \sum_{x \in \mathcal{E}} p(x)$$