Algorithms for Data Science

CSOR W4246

Eleni Drinea
Computer Science Department

Columbia University

Binary search, quicksort, randomized quicksort, occupancy problems

Outline

1. Recap
2. Binary search
3. Quicksort
4. Randomized Quicksort
5. Random variables and linearity of expectation
6. Occupancy problems

Today

1. Recap
2. Binary search
3. Quicksort
4. Randomized Quicksort
5. Random variables and linearity of expectation
6. Occupancy problems

Review of the last lecture

In the last lecture we discussed

- Asymptotic notation $(O, \Omega, \Theta, o, \omega)$
- The divide & conquer principle
  - **Divide** the problem into a number of subproblems that are smaller instances of the same problem.
  - **Conquer** the subproblems by solving them recursively.
  - **Combine** the solutions to the subproblems into the solution for the original problem.
- Application: `Mergesort`
- Solving recurrences

Mergesort

```
Mergesort (A, left, right)
   if right == left then
      return
   end if
   mid = left + ⌊(right − left)/2⌋
   Mergesort (A, left, mid)
   Mergesort (A, mid + 1, right)
   Merge (A, left, right, mid)
```

- Initial call: `Mergesort`$(A, 1, n)$
- Subroutine `Merge` merges two sorted lists of sizes $\lceil n/2 \rceil$, $\lfloor n/2 \rfloor$ into one sorted list of size n in time $\Theta(n)$.

Running time of `Mergesort`

The running time of `Mergesort` satisfies:

$$T(n) = 2T(n/2) + cn, \text{ for } n \geq 2, \text{ constant } c > 0$$
$$T(1) = c$$

This structure is typical of recurrence relations:

- an *inequality* or *equation* bounds $T(n)$ in terms of an expression involving $T(m)$ for $m < n$
- a base case generally says that $T(n)$ is constant for small constant n

Remarks

- We ignore floor and ceiling notations
- A recurrence does **not** provide an asymptotic bound for $T(n)$: to this end, we must solve the recurrence

Solving recurrences, method 1: recursion trees

The technique consists of three steps

1. Analyze the first few levels of the tree of recursive calls
2. Identify a pattern
3. Sum over all levels of recursion

Example: analysis of running time of `Mergesort`

$T(n) = 2T(n/2) + cn, n \geq 2$

$T(1) = c$

A frequently occurring recurrence and its solution

The running time of many recursive algorithms is given by

$$T(n) = aT\left(\frac{n}{b}\right) + cn^k, \quad \text{for } a, c > 0,\ b > 1,\ k \geq 0$$

What is the recursion tree for this recurrence?

- a is the branching factor
- b is the factor by which the size of each subproblem shrinks

$\Rightarrow$ at level i, there are a^i subproblems, each of size n/b^i

$\Rightarrow$ each subproblem at level i requires $c(n/b^i)^k$ work

- the height of the tree is $\log_b n$ levels

$\Rightarrow$ Total work: $\sum_{i=0}^{\log_b n} a^i c(n/b^i)^k = cn^k \sum_{i=0}^{\log_b n} \left(\frac{a}{b^k}\right)^i$

Solving recurrences, method 2: Master theorem

Theorem 1 (Master theorem).

If $T(n) = aT(\lceil n/b \rceil) + O(n^k)$ *for some constants* $a > 0$, $b > 1$, $k \geq 0$, *then*

$$T(n) = \begin{cases} O(n^{\log_b a}) & , \textit{ if } a > b^k \\ O(n^k \log n) & , \textit{ if } a = b^k \\ O(n^k) & , \textit{ if } a < b^k \end{cases}$$

Example: running time of `Mergesort`

- $T(n) = 2T(n/2) + cn$:
  $a = 2, b = 2, k = 1, b^k = 2 = a \Rightarrow T(n) = O(n \log n)$

Today

1. Recap
2. Binary search
3. Quicksort
4. Randomized Quicksort
5. Random variables and linearity of expectation
6. Occupancy problems

Searching a sorted array

- **Input:**
  1. sorted list A of n integers;
  2. integer x
- **Output:**
  - index j such that $1 \leq j \leq n$ and $A[j] = x$; *or*
  - **no** if x is not in A

Searching a sorted array

- **Input:**
  1. sorted list A of n integers;
  2. integer x
- **Output:**
  - index j such that $1 \leq j \leq n$ and $A[j] = x$; *or*
  - **no** if x is not in A

Example: $A = \{0, 2, 3, 5, 6, 7, 9, 11, 13\}$, $n = 9$, $x = 7$

Searching a sorted array

- **Input:**
  1. sorted list A of n integers;
  2. integer x
- **Output:**
  - index j such that $1 \leq j \leq n$ and $A[j] = x$; *or*
  - **no** if x is not in A

Example: $A = \{0, 2, 3, 5, 6, 7, 9, 11, 13\}$, $n = 9$, $x = 7$

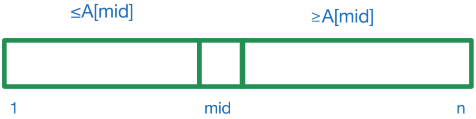
Idea: use the fact that the array is sorted and probe specific entries in the array.

Binary search

First, probe the middle entry. Let $mid = \lceil n/2 \rceil$.

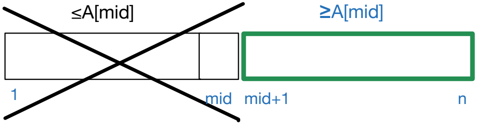
- If $x == A[mid]$, return mid.
- If $x < A[mid]$ then look for x in $A[1, mid-1]$;
- Else if $x > A[mid]$ look for x in $A[mid+1, n]$.

Initially, the entire array is “active”, that is, x might be anywhere in the array.



Suppose x > A[mid].

Then the active area of the array, where x might be, is to the right of mid.



Binary search pseudocode

```
binarysearch(A, left, right)
    mid = left + ⌈(right − left)/2⌉
    if x == A[mid] then
        return mid
    else if right == left then
        return no
    else if x > A[mid]  then
        left = mid + 1
    else right = mid − 1
    end if
    binarysearch(A, left, right)
```

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

Binary search running time

Observation: At each step there is a region of A where x could be and we **shrink** the size of this region by a factor of 2 with every probe:

- If n is odd, then we are throwing away $\lceil n/2 \rceil$ elements.
- If n is even, then we are throwing away at least $n/2$ elements.

Binary search running time

Observation: At each step there is a region of A where x could be and we **shrink** the size of this region by a factor of 2 with every probe:

- If n is odd, then we are throwing away $\lceil n/2 \rceil$ elements.
- If n is even, then we are throwing away at least $n/2$ elements.

Hence the recurrence for the running time is

$$T(n) \leq T(n/2) + O(1)$$

Sublinear running time

Here are two ways to argue about the running time:

1. Master theorem: $b = 2, a = 1, k = 0 \Rightarrow T(n) = O(\log n)$.

2. We can reason as follows: starting with an array of size n,

   - After k probes, the array has size at most $\frac{n}{2^k}$ (every time we probe an entry, the active portion of the array halves).
   - After $k = \log n$ probes, the array has **constant** size. We can now search **linearly** for x in the constant size array.
   - We spend **constant** work to halve the array (*why?*). Thus the total work spent is $O(\log n)$.

Concluding remarks on binary search

1. The right data structure can improve the running time of the algorithm significantly.
   - *What if we used a* **linked list** *to store the input?*
   - Arrays allow for **random access** of their elements: given an index, we can read any entry in an array in time $O(1)$ (constant time).

2. In general, we obtain running time $O(\log n)$ when the algorithm does a **constant amount of work** to throw away a **constant fraction** of the input.

Today

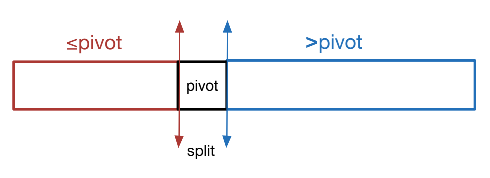
1. Recap
2. Binary search
3. Quicksort
4. Randomized Quicksort
5. Random variables and linearity of expectation
6. Occupancy problems

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

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*.

Quicksort pseudocode

```
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)
```

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

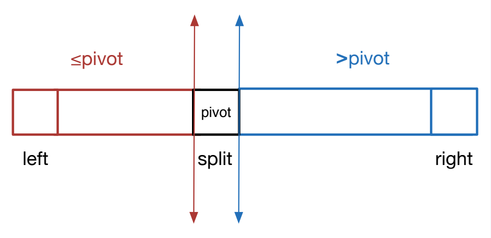
Subroutine $\texttt{Partition}(A, left, right)$

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

$\texttt{Partition}(A, left, right)$

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

After $\texttt{Partition}$, $A[left, right]$ looks as follows:



Implementing `Partition`

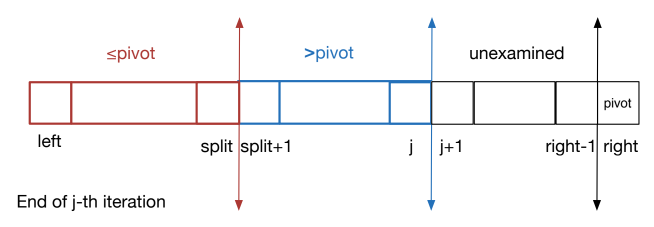
1. Pick a *pivot* item: for simplicity, always pick the last item of the array as *pivot*, i.e., $pivot = A[right]$.
   - Thus $A[right]$ will be placed in its final location in the sorted output when `Partition` returns; it will never be used (or moved) again until the algorithm terminates.
2. Re-organize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

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

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.



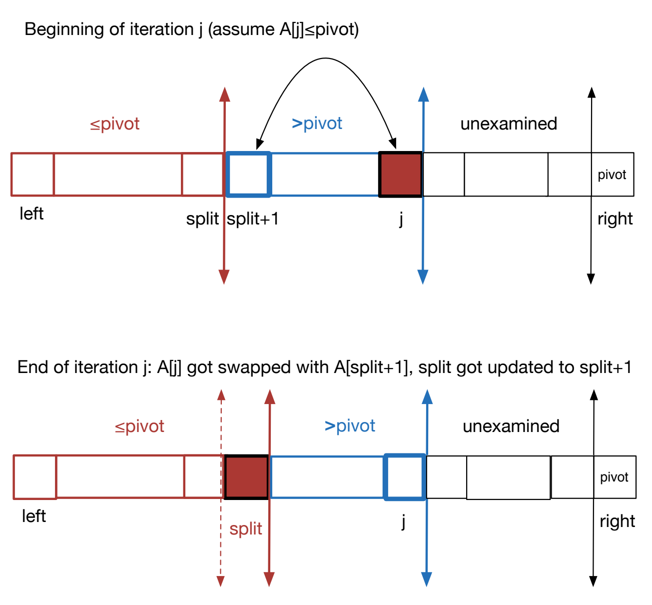
Implementing `Partition` in place (cont'd)

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

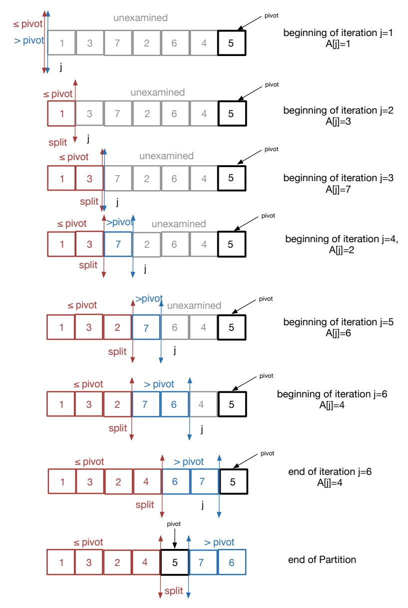
If $A[j] \leq pivot$

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



Pseudocode for Partition

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

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?)*.

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: $A = [1 \;\; 3 \;\; 2 \quad 5 \;\; 7 \;\; 6 \quad 4]$

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)

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.

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 Quicksort

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

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

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)
```

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

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*?

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 2.

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*?

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 2, $E[X_i] = p$, 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 2, $E[X_i] = p$, 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

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*

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

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

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

Application: the birthday paradox

Use $1+x \leq e^x$ for all $x \geq 0$ to upper bound (1)

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

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?*
- *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)?*

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.

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

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

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.

□

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 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; $\Omega = \{heads, tails\}$; $\Pr[heads] = \Pr[tails] = 1/2$.
- 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,

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