Randomly permuting arrays

- We assume that we are given an array \( A \) which, w.l.g., contains the elements \( 1, 2, \ldots, n \).
- Produce a random permutation of the array.
- Assign element \( A[i] \) a random priority \( P[i] \), and sort the elements according to priorities.
- E.g., if our initial array is \( A = \langle 1, 2, 3, 4 \rangle \) and we choose random priorities \( P = \langle 36, 3, 62, 19 \rangle \), we would produce array \( B = \langle 2, 4, 1, 3 \rangle \).
- We call this procedure PERMUTE-BY-SORTING.

**Permute-By-Sorting(A)**

1. \( n = A.length \)
2. let \( P[1..n] \) be a new array
3. for \( i = 1 \) to \( n \)
4. \( P[i] = \text{RANDOM}(1, n^3) \)
5. sort \( A \), using \( P \) as sort keys
- We use a range of \( 1 \) to \( n^3 \) for random numbers to make it likely that all the priorities in \( P \) are unique.
• It remains to prove that the procedure produces a *uniform random permutation*,
  – It is equally likely to produce every permutation of the numbers $1$ through $n$

**Lemma 5.4**  *PERMUTE-BY-SORTING* produces a uniform random permutation of the input, assuming that all priorities are distinct

**Proof**  See the book.

• It is better to permute the given array in place
• *RANDOMIZE-IN-PLACE* does so in $O(n)$ time
• In its $i$th iteration, it chooses the element $A[i]$ randomly from among elements $A[i]$ through $A[n]$
• After the $i$th iteration, $A[i]$ is never altered

**RANDOMIZE-IN-PLACE**(A)
1  $n = A\. length$
2  for $i = 1$ to $n$
3  swap $A[i]$ with $A[RANDOM(i, n)]$
A $k$-permutation on a set of $n$ elements is a sequence containing $k$ of the $n$ elements, with no repetitions.

There are $n!/(n-k)!$ $k$-permutations.

**Loop invariant:**

Just prior to the $i$th iteration of the for loop of lines 2–3, for each possible $(i-1)$-permutation of the $n$ elements, the subarray $A[1..i-1]$ contains this $(i-1)$-permutation with probability $(n-i+1)!/n!$.

**Initialization:** loop invariant trivially holds

**Maintenance:**

- Consider a particular $i$-permutation, and denote the elements in it by $\langle x_1, x_2, \ldots, x_i \rangle$.
- This permutation consists of an $(i-1)$-permutation $\langle x_1, x_2, \ldots, x_{i-1} \rangle$ followed by the value $x_i$ that the algorithm places in $A[i]$.
- Let $E_1$ denote the event in which the first $(i-1)$ iterations have created the particular $(i-1)$-permutation $\langle x_1, x_2, \ldots, x_{i-1} \rangle$ in $A[1..i-1]$. 
• By the loop invariant, $Pr\{E_1\} = (n - i + 1)!/n!$

• Let $E_2$ be the event that $i$th iteration puts $x_i$ in position $A[i]$

• The $i$-permutation $(x_1, x_2, \ldots, x_i)$ appears in $A[1..i]$ precisely when both $E_1$ and $E_2$ occur

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

• $Pr\{E_2 | E_1\} = 1/(n - i + 1)$ because in line 3 the algorithm chooses $x_i$ randomly from the $n - i + 1$ values in positions $A[i..n]$

$$Pr\{E_2 \cap E_1\} = \frac{1}{n - i + 1} \cdot \frac{(n - i + 1)!}{n!} = \frac{(n - i)!}{n!}$$

**Termination:**

• At termination, $i = n + 1$, and we have that the subarray $A[1..n]$ is a given $n$-permutation with probability

$$(n - (n + 1) + 1)!/n! = 0!/n! = 1/n!$$

• Thus, RANDOMIZE-IN-PLACE produces a uniform random permutation
5.4.1 The birthday paradox

• How many people must there be in a room before there is a 50% chance that two of them were born on the same day of the year?

• The answer is surprisingly few
• The paradox is that it is in fact far fewer
  – than the number of days in a year, or
  – even half the number of days in a year

An analysis using indicator random variables

• We use indicator random variables to provide a simple but approximate analysis of the birthday paradox
• For each pair \((i, j)\) of the \(k\) people in the room, define the indicator random variable \(X_{ij}\), for \(1 \leq i < j \leq k\), by

\[
X_{ij} = \begin{cases} 
1 & \text{if } i \text{ and } j \text{ have the same birthday} \\
0 & \text{otherwise}
\end{cases}
\]
Once birthday \( b_i \) for \( i \) is chosen, the probability that \( b_j \) is chosen to be the same day is \( 1/n \), where \( n = 365 \).

\[
E[X_{ij}] = \Pr \{ i \text{ and } j \text{ have the same birthday} \} = 1/n
\]

Let \( X \) be a random variable counting the number of pairs of individuals having the same birthday

\[
X = \sum_{i=1}^{k} \sum_{j=i+1}^{k} X_{ij}
\]

Taking expectations of both sides and applying linearity of expectation, we obtain

\[
E[X] = E \left[ \sum_{i=1}^{k} \sum_{j=i+1}^{k} X_{ij} \right]
= \sum_{i=1}^{k} \sum_{j=i+1}^{k} E[X_{ij}]
= \binom{k}{2} \frac{1}{n} = \frac{k(k-1)}{2n}
\]

When \( k(k-1) \geq 2n \), the expected number of pairs of people with the same birthday is at least 1.
Thus, if we have at least $\sqrt{2n} + 1$ individuals in a room, we can expect at least two to have the same birthday.

For $n = 365$, if $k = 28$, the expected number of pairs with the same birthday is $(28 \cdot 27)/(2 \cdot 365) \approx 1.0356$.

With at least 28 people, we expect to find at least one matching pair of birthdays.

Analysis using only probabilities gives a different exact number of people, but same asymptotically: $\Theta(\sqrt{n})$.

### 5.4.2 Balls and bins

Consider tossing identical balls randomly into $b$ bins, numbered $1,2,\ldots,b$.

Tosses are independent, and on each toss the ball is equally likely to end up in any bin.

The probability that a tossed ball lands in any given bin is $1/b$.

The ball-tossing process is a sequence of Bernoulli trials with a probability $1/b$ of success $\equiv$ the ball falls in the given bin.
• **How many balls fall in a given bin?**
  – The number of balls that fall in a given bin follows the binomial distribution \( b(k; n, 1/b) \)
  – If we toss \( n \) balls, the expected number of balls that fall in the given bin is \( n/b \)

• **How many balls must we toss, on the average, until a given bin contains a ball?**
  – The number of tosses until the given bin receives a ball follows the geometric distribution with probability \( 1/b \) and
  – the expected number of tosses until success is \( 1/(1/b) = b \)

• **How many balls must we toss until every bin contains at least one ball?**
  – Call a toss in which a ball falls into an empty bin a “hit”
  – We want to know the expected number \( n \) of tosses required to get \( b \) hits
  – We can partition the \( n \) tosses into stages
  – The \( i \)th stage consists of the tosses after the \( (i - 1) \)st hit until the \( i \)th hit
  – The first stage consists of the first toss, since we are guaranteed to have a hit when all bins are empty
- During the $i$th stage, $i - 1$ bins contain balls and $b - i + 1$ bins are empty.
- For each toss in the $i$th stage, the probability of obtaining a hit is $(b - i + 1)/b$.
- $n_i$ is the number of tosses in the $i$th stage.
- The number of tosses required to get $b$ hits is $n = \sum_{i=1}^{b} n_i$.
- Each $n_i$ has a geometric distribution with probability of success $(b - i + 1)/b$.

$$E[n_i] = \frac{b}{b - i + 1}$$

$$E[n] = E\left[\sum_{i=1}^{b} n_i\right] = \sum_{i=1}^{b} E[n_i]$$

$$= \sum_{i=1}^{b} \frac{b}{b - i + 1}$$

$$= b \sum_{i=1}^{b} \frac{1}{i}$$

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

- By harmonic series.
It therefore takes approximately $b \ln b$ tosses before we can expect that every bin has a ball.

This problem is also known as the **coupon collector’s problem**, which says that a person trying to collect each of $b$ different coupons expects to acquire approximately $b \ln b$ randomly obtained coupons in order to succeed.
6 Heapsort

- Heapsort’s running time is $O(n \lg n)$
- It sorts in place
  - only a constant number of array elements are stored outside the input array at any time
- Heapsort combines the better attributes of the sorting algorithms we have already discussed
- Heapsort also introduces another algorithm design technique: using a data structure

6.1 Heaps

- The (binary) heap data structure is an array object that we can view as a nearly complete binary tree
- Each node of the tree corresponds to an element of the array
- The tree is completely filled on all levels except possibly the lowest, which is filled from the left up to a point
An array $A$ that represents a heap is an object with two attributes:

- $A.\text{length}$ gives the number of elements in the array, and
- $A.\text{heap-size}$ represents how many elements in the heap are stored within array $A$

Although $A[1..A.\text{length}]$ may contain numbers, only the elements in $A[1..A.\text{heap-size}]$, where $0 \leq A.\text{heap-size} \leq A.\text{length}$, are valid elements of the heap.

The root of the tree is $A[1]$, and given the index $i$ of a node, we can easily compute the indices of its parent, left child, and right child:

- $\text{PARENT}(i) \equiv \text{return } [i/2]$
- $\text{LEFT}(i) \equiv \text{return } 2i$
- $\text{RIGHT}(i) \equiv \text{return } 2i + 1$
There are two kinds of binary heaps:
- max-heaps and min-heaps

In both, the values in the nodes satisfy a **heap property**

- The max-heap property is that for every node \( i \) other than the root
  \[
  A[\text{PARENT}(i)] \geq A[i]
  \]
- The largest element in a max-heap is stored at the root
- The subtree rooted at node \( n \) contains values no larger than that contained at \( n \) itself

The **height** of a node in a heap is the number of edges on the longest simple downward path from the node to a leaf, and
- We define the height of the heap to be the height of its root
- Since a heap of \( n \) elements is based on a complete binary tree, its height is \( \Theta(\lg n) \)
- The basic operations on heaps run in time at most proportional to the height of the tree and thus take \( O(\lg n) \) time
6.2 Maintaining the heap property

- MAX-HEAPIFY maintains the heap property
- Its inputs are an array $A$ and index $i$ into it
- It assumes that the binary trees rooted at $\text{LEFT}(i)$ and $\text{RIGHT}(i)$ are max-heaps, but that $A[i]$ might be smaller than its children
- MAX-HEAPIFY lets the value at $A[i]$ “float down” in the max-heap so that the subtree rooted at index $i$ obeys the max-heap property

MAX-HEAPIFY($A, i$)

1. $l = \text{LEFT}(i)$
2. $r = \text{RIGHT}(i)$
3. if $l \leq A.\text{heap-size}$ and $A[l] > A[i]$
   4. $\text{largest} = l$
5. else $\text{largest} = i$
6. if $r \leq A.\text{heap-size}$ and $A[r] > A[\text{largest}]$
   7. $\text{largest} = r$
8. if $\text{largest} \neq i$
9. exchange $A[i]$ with $A[\text{largest}]$
10. MAX-HEAPIFY($A, \text{largest}$)
At each step, the largest of the elements \( A[i] \), \( A[\text{LEFT}(i)] \), and \( A[\text{RIGHT}(i)] \) is determined, and its index is stored in \textit{largest}.

- If \( A[i] \) is largest, then the subtree rooted at node \( i \) is already a max-heap.
- Otherwise, one child has the largest element, and \( A[i] \) is swapped with \( A[\text{largest}] \).
- The node indexed by \textit{largest} now has the original value \( A[i] \), and thus the subtree rooted at \textit{largest} might violate the max-heap property => call \textsc{MAX-HEAPIFY} recursively.
• Running time of MAX-HEAPIFY on a subtree of size $n$ rooted at a given node $i$ is
  – the $\Theta(1)$ time to fix up the relationships of $A[i], A[\text{LEFT}(i)],$ and $A[\text{RIGHT}(i)]$
  – plus the time to run MAX-HEAPIFY on a subtree rooted at one of the children of node $i$
• The children’s subtrees have size at most $\frac{2n}{3}$
  • the worst case occurs when the bottom level of the tree is exactly half full
• We have the recurrence $T(n) = T(2n/3) + \Theta(1) = O(\lg n)$

6.3 Building a heap

• We can use the procedure MAX-HEAPIFY bottom-up to convert an array $A[1..n]$, where $n = A.length$, into a max-heap
• Elements in the subarray $A[(\lfloor n/2 \rfloor + 1)..n]$ are all leaves of the tree, and so each is a 1-element heap to begin with
• BUILD-MAX-HEAP goes through the remaining nodes of the tree and runs MAX-HEAPIFY on each one
**BUILD-MAX-HEAP(\(A\))**

1. \(A.\text{heap-size} = A.\text{length}\)
2. **for** \(i = \lfloor A.\text{length}/2 \rfloor \textbf{ downto } 1\)
3. **MAX-HEAPIFY(\(A, i\))**

- A simple upper bound on the running time:
  - Each call to **MAX-HEAPIFY** costs \(O(\lg n)\) time
  - **BUILD-MAX-HEAP** makes \(O(n)\) such calls
  - Thus, the running time is \(O(n \lg n)\)
- This upper bound is not asymptotically tight
• Observe that the time for MAX-HEAPIFY to run at a node varies with the height of the node in the tree
• The heights of most nodes are small
• The time required by MAX-HEAPIFY when called on a node of height $h$ is $O(h)$
• The total cost of BUILD-MAX-HEAP is bounded by $O(n)$
• Hence, we can build a max-heap from an unordered array in linear time

6.4 The heapsort algorithm

• The heapsort algorithm starts by using BUILD-MAX-HEAP to build a max-heap on the input array $A[1..n]$
• The maximum element of the array is stored at the root $A[1]$, we can put it into its correct final position by exchanging it with $A[n]$
• We can now discard node $n$ from the heap by simply decrementing $A.heap-size$
• The children of the root remain max-heaps, but the new root element might violate the max-heap property
• To restore the max-heap property call MAX-HEAPIFY\((A,1)\), which leaves a max-heap in 
\(A[1..n-1]\)
• The heapsort algorithm then repeats this process for the max-heap of size \(n-1\) down to a heap of size 2

HEAPSORT\((A)\)
1. BUILD-MAX-HEAP\((A)\)
2. for \(i = A.\text{length} \) downto 2
3. exchange \(A[1]\) with \(A[i]\)
4. \(A.\text{heap-size} = A.\text{heap-size} - 1\)
5. MAX-HEAPIFY\((A,1)\)

• The procedure takes time \(O(n \lg n)\), since the call to BUILD-MAX-HEAP takes time \(O(n)\) and each of the \(n-1\) calls to MAX-HEAPIFY takes time \(O(\lg n)\)
6.5 Priority queues

- Maintain a set $S$ of elements, each with an associated value called a **key**
- **INSERT**$(S, x)$ insert $x$ into $S$: $S = S \cup \{x\}$
- **MAXIMUM**$(S)$ return element with largest key
- **EXTRACT-MAX**$(S)$ removes and returns the element of $S$ with the largest key
- **INCREASE-KEY**$(S, x, k)$ increases the value of element $x$’s key to the new value $k$, which is at least as large as $x$’s current key value

We can use max-priority queues, e.g., to schedule jobs on a shared computer
- The max-priority queue keeps track of the jobs to be performed and their relative priorities
- When a job is finished or interrupted, the scheduler selects the highest-priority job from among pending ones by calling **EXTRACT-MAX**
- The scheduler can add a new job to the queue at any time by calling **INSERT**
• It is easy to implement \textsc{Maximum} operation in $\Theta(1)$ time using a heap

\begin{verbatim}
\textsc{Heap-Maximum}(A) \equiv \text{return } A[1]
\end{verbatim}

\textsc{Heap-Extract-Max}(A)

1. \text{if } A.\text{heap-size} < 1
2. \text{error “heap underflow”}
3. \text{max = } A[1]
5. \text{ } A.\text{heap-size} = A.\text{heap-size} - 1
6. \text{ } \textsc{Max-Heapify}(A, 1)
7. \text{return } \text{max}

• Running time of \textsc{Heap-Extract-Max} is $O(\lg n)$; it performs a constant amount of work on top of the $O(\lg n)$ for \textsc{Max-Heapify}

• In \textsc{Heap-Increase-Key} index $i$ identifies the element whose key we wish to increase
• Increasing the key of $A[i]$ might violate the max-heap property
• The procedure traverses a simple path from this node toward the root to find a proper place for the newly increased key
The procedure repeatedly compares an element to its parent, exchanging their keys and continuing if the element’s key is larger, and terminating otherwise.

**HEAP-INCREASE-KEY** \((A, i, key)\)

1. If \(key < A[i]\)
2. **error** “new key is smaller than current key”
3. \(A[i] = key\)
4. **while** \(i > 1\) **and** \(A[\text{PARENT}(i)] < A[i]\)
5. exchange \(A[i]\) with \(A[\text{PARENT}(i)]\)
6. \(i = \text{PARENT}(i)\)
• The running time of \textsc{Heap-Increase-Key} on an \( n \)-element heap is \( O(\lg n) \), since the path traced to the root has length \( O(\lg n) \)

\textsc{Max-Heap-Insert}(\( A, key \))
1. \( A.\text{heap-size} = A.\text{heap-size} + 1 \)
2. \( A[A.\text{heap-size}] = -\infty \)
3. \textsc{Heap-Increase-Key} \( (A, A.\text{heap-size}, key) \)

• Also this procedure has running time \( O(\lg n) \)

7 Quicksort

• Quicksort has a worst-case running time of \( \Theta(n^2) \) on an array of \( n \) numbers

• Despite this, quicksort is often the best practical choice for sorting because it is remarkably efficient on the average:
  – Its expected running time is \( \Theta(n \lg n) \), and the constant factors are quite small

• It also sorts in place and it works well even in virtual-memory environments
7.1 Description of quicksort

- **Divide**: Rearrange the array \( A[p..r] \) into two (possibly empty) subarrays \( A[p..q-1] \) and \( A[q+1..r] \)
  - such that each element of \( A[p..q-1] \leq A[q] \), which in turn \( \leq \) each element of \( A[q+1..r] \)
  
  Compute the index \( q \) as part of this partitioning procedure

- **Conquer**: Sort the two subarrays by recursive calls to quicksort

- **Combine**: Because the subarrays are already sorted, no work is needed to combine them: the entire array \( A[p..r] \) is now sorted

**QUICKSORT**\( (A, p, r) \)

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

• To sort an entire array \( A \), the initial call is **QUICKSORT**\( (A, 1, A.length) \)
Partitioning the array

\textbf{PARTITION}(A, p, r)

1. \( x = A[r] \)
2. \( i = p - 1 \)
3. \textbf{for} \( j = p \) \textbf{to} \( r - 1 \)
4. \textbf{if} \( A[j] \leq x \)
5. \( i = i + 1 \)
7. \textbf{exchange} \( A[i + 1] \) \textbf{with} \( A[r] \)
8. \textbf{return} \( i + 1 \)
• **PARTITION** always selects an element \( x = A[r] \) as a *pivot* element around which to partition the subarray \( A[p..r] \).

• As the procedure runs, it partitions the array into four (possibly empty) regions.

• At the start of each iteration of the **for** loop in lines 3–6, the regions satisfy properties, shown above.

• At the beginning of each iteration of the loop of lines 3–6, for any array index \( k \),
  1. If \( p \leq k \leq i \), then \( A[k] \leq x \)
  2. If \( i + 1 \leq k \leq j - 1 \), then \( A[k] > x \)
  3. If \( k = r \), then \( A[k] = x \)

• Indices between \( j \) and \( r - 1 \) are not covered by any case, and the values in these entries have no particular relationship to the pivot \( x \).

• The running time of **PARTITION** on the subarray \( A[p..r] \) is \( \Theta(n) \), \( n = p - r + 1 \).
7.2 Performance of quicksort

- Running time of quicksort depends on whether the partitioning is balanced or unbalanced
  - which in turn depends on which elements are used for partitioning.
  - Balanced: the algorithm runs asymptotically as fast as merge sort
  - Unbalanced: it can run asymptotically as slowly as insertion sort

Worst-case partitioning

- The worst-case behavior occurs when the partitioning routine produces one subproblem with \( n - 1 \) elements and one with 0 elements
- Let us assume that this unbalanced partitioning arises in each recursive call
- Partitioning costs \( \Theta(n) \), time
- The recursive call on an array of size 0 just returns, \( T(0) = \Theta(1) \)
The recurrence for the running time is
\[ T(n) = T(n - 1) + T(0) + \Theta(n) \]
\[ = T(n - 1) + \Theta(n) \]
- The substitution method proves that this recurrence has the solution \( T(n) = \Theta(n^2) \)
- The worst-case running time of quicksort is no better than that of insertion sort
- The \( \Theta(n^2) \) running time occurs when the input array is already completely sorted
  – a common situation in which insertion sort runs in \( O(n) \) time

Best-case partitioning
- In the most even possible split, \textsc{partition} produces two subproblems, each of size at most \( n/2 \): of sizes \( [n/2] \) and \( [n/2] - 1 \)
- The recurrence for the running time is then
  \[ T(n) = 2T(n/2) + \Theta(n) \]
- By case 2 of the master theorem, this recurrence has the solution \( T(n) = \Theta(n \log n) \)
- By balancing the two sides of the partition, we get an asymptotically faster algorithm
Balanced partitioning

- Average-case running time of quicksort is closer to the best case than to the worst case
- Suppose, that the partitioning algorithm always produces a 9-to-1 proportional split – at first blush this seems quite unbalanced
- We then obtain the recurrence
  \[ T(n) = T(9n/10) + T(n/10) + cn, \]
  where we have explicitly included the constant \( c \) hidden in the \( \Theta(n) \) term
In the recursion tree every level of the tree has cost $cn$, until the recursion reaches a boundary condition at depth $\log_{10} n = \Theta(lg n)$, and then the levels have cost $\leq cn$

- Recursion terminates at $\log_{10/9} n = \Theta(lg n)$
- Total cost of quicksort is therefore $O(n \lg n)$
- Thus, with this intuitively quite unbalanced seeming split quicksort runs asymptotically as if the split were right down the middle
- In fact, any split of constant proportionality leads to $O(n \lg n)$ running time