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 \( \frac{1}{b} \)
- The ball-tossing process is a sequence of Bernoulli trials with a probability \( \frac{1}{b} \) 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 $(i - 1)$th 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 \left( \ln b + O(1) \right)

- 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
9 Medians and Order Statistics

- The $i$th order statistic of a set of $n$ elements is the $i$th smallest element
  - E.g., the minimum of a set of elements is the first order statistic ($i = 1$), and the maximum is the $n$th order statistic ($i = n$)
- A **median** is the “halfway point” of the set
- When $n$ is odd, the median is unique, occurring at $i = (n + 1)/2$
When \( n \) is even, there are two medians, occurring at \( i = n/2 \) and \( i = n/2 + 1 \).

Thus, regardless of the parity of \( n \), medians occur at:
- \( [i = (n + 1)/2] \) (the lower median) and
- \( [i = (n + 1)/2] \) (the upper median).

For simplicity, we use “the median” to refer to the lower median.

The problem of selecting the \( i \)th order statistic from a set of \( n \) distinct numbers.

We assume that the set contains distinct numbers:
- virtually everything extends to the situation in which a set contains repeated values.

We formally specify the problem as follows:
- **Input**: A set \( A \) of \( n \) (distinct) numbers and an integer \( i \), with \( 1 \leq i \leq n \).
- **Output**: The element \( x \in A \) that is larger than exactly \( i - 1 \) other elements of \( A \).
• We can solve the problem in $O(n \lg n)$ time by heapsort or merge sort and then simply index the $i$th element in the output array.

• There are faster algorithms.

• First, we examine the problem of selecting the minimum and maximum of a set of elements.

• Then we analyze a practical randomized algorithm that achieves an $O(n)$ expected running time, assuming distinct elements.

9.1 Minimum and maximum

• How many comparisons are necessary to determine the minimum of a set of $n$ elements?

• We can easily obtain an upper bound of $n - 1$ comparisons.
  – examine each element of the set in turn and keep track of the smallest element seen so far.

• In the following procedure, we assume that the set resides in array $A$, where $A.length = n$. 
\textbf{Minimum}(A)

1. \text{min} \leftarrow A[1]
2. \textbf{for} \ i \leftarrow 2 \ \textbf{to} \ A\.\text{length}
3. \textbf{if} \ \text{min} > A[i]
4. \quad \text{min} \leftarrow A[i]
5. \textbf{return} \ \text{min}

- We can, of course, find the maximum with \( n - 1 \) comparisons as well

- This is the best we can do, since we can obtain a lower bound of \( n - 1 \) comparisons
  - Think of any algorithm that determines the minimum as a tournament among the elements
  - Each comparison is a match in the tournament in which the smaller of the two elements wins
  - Observing that every element except the winner must lose at least one match, we conclude that \( n - 1 \) comparisons are necessary to determine the minimum
- Hence, the algorithm \text{Minimum} is optimal w.r.t. the number of comparisons performed
Simultaneous minimum and maximum

- Sometimes, we must find both the minimum and the maximum of a set of $n$ elements.
- For example, a graphics program may need to scale a set of $(x, y)$ data to fit onto a rectangular display screen or other graphical output device.
- To do so, the program must first determine the minimum and maximum value of each coordinate.

$\Theta(n)$ comparisons is asymptotically optimal:

- Simply find the minimum and maximum independently, using $n - 1$ comparisons for each, for a total of $2n - 2$ comparisons.
- In fact, we can find both the minimum and the maximum using at most $3 \lceil n/2 \rceil$ comparisons by maintaining both the minimum and maximum elements seen thus far.
- Rather than processing each element of the input by comparing it against the current minimum and maximum, we process elements in pairs.
• Compare pairs of input elements first with each other, and then we compare the smaller with the current min and the larger to the current max, at a cost of 3 comparisons for every 2 elements

• If \( n \) is odd, we set both the min and max to the value of the first element, and then we process the rest of the elements in pairs

• If \( n \) is even, we perform 1 comparison on the first 2 elements to determine the initial values of the min and max, and then process the rest of the elements in pairs as in the case for odd \( n \)

• If \( n \) is odd, then we perform

\[3 \left\lfloor \frac{n}{2} \right\rfloor\]

comparisons

• If \( n \) is even, we perform 1 initial comparison followed by

\[3 \left( n - 2 \right)/2\]

comparisons, for a total of \(3n/2 - 2\)

• Thus, in either case, the total number of comparisons is at most \(3\lceil n/2 \rceil\)
9.2 Selection in expected linear time

- The selection problem appears more difficult than finding a minimum, but the asymptotic running time for both is the same: $\Theta(n)$
- A divide-and-conquer algorithm RANDOMIZED-SELECT is modeled after the quicksort algorithm
- Unlike quicksort, RANDOMIZED-SELECT works on only one side of the partition
- Whereas quicksort has an expected running time of $\Theta(n \log n)$, the expected running time of RANDOMIZED-SELECT is $\Theta(n)$, assuming that the elements are distinct

RANDOMIZED-SELECT uses the procedure RANDOMIZED-PARTITION of RANDOMIZED-QUICKSORT

RANDOMIZED-PARTITION($A, p, r$)
1. $i \leftarrow $ RANDOM($p, r$)
2. exchange $A[r]$ with $A[i]$
3. return PARTITION($A, p, r$)
Partitioning the array

\textsc{Partition}(A, p, r)

1. \( x \leftarrow A[r] \)
2. \( i \leftarrow p - 1 \)
3. \( \text{for } j \leftarrow p \text{ to } r - 1 \)
4. \( \text{if } A[j] \leq x \)
5. \( i \leftarrow i + 1 \)
6. \( \text{exchange } A[i] \text{ with } A[j] \)
7. \( \text{exchange } A[i + 1] \text{ with } A[r] \)
8. \( \text{return } i + 1 \)
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 $\Theta(n)$, $n = p - r + 1$.
Return the $i$th smallest element of $A[p..r]$

**RANDOMIZED-SELECT**($A, p, r, i$)

1. if $p = r$
2. return $A[p]$
3. $q \leftarrow$ **RANDOMIZED-PARTITION**($A, p, r$)
4. $k \leftarrow q - p + 1$
5. if $i = k$ // the pivot value is the answer
6. return $A[q]$
7. elseif $i < k$
8. return **RANDOMIZED-SELECT**($A, p, q - 1, i$)
9. else return **RANDOMIZED-SELECT**($A, q + 1, r, i - k$)

(1) checks for the base case of the recursion
Otherwise, **RANDOMIZED-PARTITION** partitions $A[p..r]$ into two (possibly empty) subarrays $A[p..q - 1]$ and $A[q + 1..r]$ s.t. each element in the former is $\leq A[q] <$ each element of the latter
(4) computes the # $k$ of elements in $A[p..q]$
(5) checks if $A[q]$ is $i$th smallest element
Otherwise, determine in which of the two subarrays the $i$th smallest element lies
If $i < k$, then the desired element lies on the low side of the partition, and (8) recursively selects it
If $i > k$, then the desired element lies on the high side of the partition.

Since we already know $k$ values that are smaller than the $i$th smallest element of $A[p..r]$ the desired element is the $(i - k)$th smallest element of $A[q + 1..r]$, which (9) finds recursively.