Section 5.1 – The hiring problem

5.1-1 Show that the assumption that we are always able to determine which candidate is best, in line 4 of procedure Hire-Assistant, implies that we know a total order on the ranks of the candidates.

Let A be the set of candidates in random order and R the binary relation “is better than or equal” on the set A. R is a total order if

(a) R is **reflexive**. That is, $a\ R\ a\ \forall\, a \in A$;

(b) R is **antisymmetric**. That is, $a\ R\ b$ and $b\ R\ a$ imply $a = b$;

(c) R is **transitive**. That is, $a\ R\ b$ and $b\ R\ c$ imply $a\ R\ c$;

(d) R is a **total relation**. That is, $a\ R\ b$ or $b\ R\ a\ \forall\, a, b \in A$.

The above properties are necessary because

(a) if two different candidates have the same qualification, it is necessary so that they can be compared;

(b) if both a is “better than or equal” than b and b is “better than or equal” than a and they qualifications are not equal, we would not be able to choose one of them and still be hiring “the best candidate we have seen so far”;

(c) if we hire b because he is “better than or equal” than a and then we hire c because he is “better than or equal” than b and c is not “better than or equal” than a, we are not hiring “the best candidate we have seen so far”;

(d) if R is not a total relation, we would not be able to compare any two candidates.

5.1-2 ($\star$) Describe an implementation of the procedure Random(a, b) that only makes calls to Random(0, 1). What is the expected running time of your procedure, as a function of a and b?

The pseudocode is stated below.

```
RandomInterval(a, b)
    $flips = \lceil \lg(b-a) \rceil$
    $count = \infty$
    while $count > b$ do
        $count = 0$
        for $i = 1$ to $flips$ do
            $count = count + (2^{i-1} \cdot \texttt{Random}(0,1))$
    return $count + a$
```

The expected running time is

$$\underbrace{2^{\lceil \lg(b-a) \rceil}/(b-a)}_{\text{while loop}} \cdot \underbrace{\lceil \lg(b-a) \rceil}_{\text{for loop}} < 2 \cdot \lceil \lg(b-a) \rceil,$$

where the last inequality is valid since $1 \leq 2^{\lceil \lg(b-a) \rceil}/(b-a) < 2$.

5.1-3 ($\star$) Suppose that you want to output 0 with probability 1/2 and 1 with probability 1/2. At your disposal is a procedure Biased-Random, that outputs either 0 or 1. It outputs 1 with some probability p and 0 with probability $1-p$, where $0 < p < 1$, but you do not know what p is. Give an algorithm that uses Biased-Random as a subroutine, and returns an unbiased answer, returning 0 with probability 1/2 and 1 with probability 1/2. What is the expected running time of your algorithm as a function of p?

The pseudocode is stated below.

```
Random()
    while 1 do
        $r_1 = \texttt{Random}(0,1)$
        $r_2 = \texttt{Random}(0,1)$
        if $r_1 \neq r_2$ then
            return $r_1$
```

The expected running time is

$$\frac{1}{\underbrace{(1-p)p}_{(r_1,r_2)=(0,1)} + \underbrace{p(1-p)}_{(r_1,r_2)=(1,0)}} \cdot 1 = \frac{1}{2p(1-p)}.$$

Section 5.2 – Indicator random variables

5.2-1 In HIRE-ASSISTANT, assuming that the candidates are presented in a random order, what is the probability that you hire exactly one time? What is the probability that you hire exactly n times?

Since the initial dummy candidate is the least qualified, HIRE-ASSISTANT will always hire the first candidate. It hires exacly one time when the best candidate is the first to be interviewed. Thus, the probability is $1/n$. To hire exactly n times, the candidates has to be in increasing order of quality. Since there are $n!$ possible orderings (each one with equal probability of happening), the probability is $1/n!$.

5.2-2 In HIRE-ASSISTANT, assuming that the candidates are presented in a random order, what is the probability that you hire exactly twice?

The first candidate is always hired, thus the best qualified candidate cannot be the first to be interviewed. Also, among all the candidates that are better qualified than the first candidate, the best candidate must be interviewed first. Otherwise, a third candidate will be hired between them. Now assume that the first candidate to be interviewed is the ith best qualified, for $i = 2, \dots, n$. This occurs with a probability of $1/n$. To hire exactly twice, the best candidate must be the first to be interviewed among the $i-1$ candidates that are better qualified than candidate i. This occurs with a probability of $1/(i-1)$. Thus, the probability of hiring exactly twice is

$$\sum_{i=2}^{n} \frac{1}{n}\frac{1}{i-1} = \frac{1}{n}\sum_{i=1}^{n-1}\frac{1}{i} = \frac{1}{n}(\lg(n-1) + O(1)).$$

5.2-3 Use indicator random variables to compute the expected value of the sum of n dice.

Let X_i be an indicator random variable of a dice coming up the number i. We have $\Pr\{X_i\} = 1/6$. Let X be a random variable denoting the result of throwing a dice. Then

$$\mathrm{E}[X] = \sum_{i=1}^{6} i \cdot \Pr\{X_i\} = \sum_{i=1}^{6} i \cdot \frac{1}{6} = \frac{1}{6}\sum_{i=1}^{6} i = \frac{1}{6}\frac{6 \cdot 7}{2} = 3.5.$$

By linearity of expectations, the expected value of the sum of n dice is the sum of the expected value of each dice. Thus,

$$\sum_{i=1}^{n} \mathrm{E}[X] = \sum_{i=1}^{n} 3.5 = 3.5 \cdot n.$$

5.2-4 Use indicator random variables to solve the following problem, which is known as the ***hat-check problem***. Each of n customers gives a hat to a hat-check person at a restaurant. The hat-check person gives the hats back to the customers in a random order. What is the expected number of customers who get back their own hat?

Let X_i be an indicator random variable of customer i getting back his own hat. We have

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

Let X be a random variable denoting the number of customers who get back their own hat. Then

$$X = X_1 + X_2 + \cdots + X_n,$$

which implies

$$\begin{aligned}
\mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{n} X_i\right] \\
&= \sum_{i=1}^{n} \mathrm{E}[X_i] \\
&= \sum_{i=1}^{n} \frac{1}{n} \\
&= 1.
\end{aligned}$$

5.2-5 Let $A[1,\dots,n]$ be an array of n distinct numbers. If $i < j$ and $A[i] > A[j]$, then the pair (i,j) is called an ***inversion*** of A. (See Problem 2-4 for more on inversions.) Suppose that the elements of A form a uniform random permutation of $\langle 1,2,\dots,n\rangle$. Use indicator random variables to compute the expected number of inversions.

Let X_{ij} be an indicator random variable for the event that the pair (i,j) is inverted. Since A forms a uniform random permutation, we have

$$\Pr\{X_{ij}\} = \Pr\{\overline{X_{ij}}\} = 1/2,$$

which implies

$$\mathrm{E}[X_{ij}] = 1/2.$$

Let X be a random variable denoting the number of inversions of A. Since there are $\binom{n}{2}$ possible pairs on A, each with probability $1/2$ of being inverted, we have

$$\mathrm{E}[X] = \binom{n}{2}\frac{1}{2} = \frac{n!}{2!\cdot(n-2)!}\frac{1}{2} = \frac{n(n-1)}{4}.$$

Section 5.3 – Randomized algorithms

5.3-1 Professor Marceau objects to the loop invariant used in the proof of Lemma 5.5. He questions whether it is true prior to the first iteration. He reasons that we could just as easily declare that an empty subarray contains no 0-permutations. Therefore, the probability that an empty subarray contains a 0-permutation should be 0, thus invalidating the loop invariant prior to the first iteration. Rewrite the procedure RANDOMIZE-IN-PLACE so that its associated loop invariant applies to a nonempty subarray prior to the first iteration, and modify the proof of Lemma 5.5 for your procedure.

Just select a random element in the array and swap it with the first element.

```
Randomize-In-Place($A$)
1   $n = A.length$
2   swap $A[1]$ with $A[$Random$(1, n)]$
3   for $i = 2$ to $n - 1$ do
4       swap $A[i]$ with $A[$Random$(i, n)]$
```

The only difference in the proof of Lemma 5.5 is the initialization of the loop invariant:

- **Initialization.** Consider the situation just before the first loop iteration, so that $i = 2$. The loop invariant says that for each possible 1-permutation, the subarray $A[1, \dots, 1]$ contains this 1-permutation with probability $(n-i+1)/n! = (n-1)!/n! = 1/n$. The subarray $A[1, \dots, 1]$ has a single element and this element was randomly choosed among the n elements of the array. Thus, $A[1, \dots, 1]$ contains this 1-permutation with probability $1/n$, and the loop invariant holds prior to the first iteration.

5.3-2 Professor Kelp decides to write a procedure that produces at random any permutation besides the identity permutation. He proposes the following procedure:

```
Permute-Without-Identity($A$)
1   $n = A.length$
2   for $i = 1$ to $n - 1$ do
3       swap $A[i]$ with $A[$Random$(i + 1, n)]$
```

Does this code do what Professor Kelp intends?

No. This code enforces that every position i of the resulting array receives an element that is different from the ith element of the original array. However, this requirement discards much more permutations than just the identity permutation. For instance, consider the array $A = [1, 2, 3]$ and a permutation of it $A' = [1, 3, 2]$. In this case, the permutation A' is not identical to the original array A. However, Professor Kelp's code is not able to produce this permutation.

5.3-3 Suppose that instead of swapping element $A[i]$ with a random element from the subarray $A[i, \dots, n]$, we swapped it with a random element from anywhere in the array:

```
Permute-With-All($A$)
1   $n = A.length$
2   for $i = 1$ to $n$ do
3       swap $A[i]$ with $A[$Random$(1, n)]$
```

Does this code produce a uniform random permutation? Why or why not?

No. As a counterexample, consider the input array $A = [1, 2, 3]$. Since each call to RANDOM can produce one of three values, the number of possible outcomes after all the RANDOM calls can be seen as the number of strings over the set $\{1, 2, 3\}$, which is $3^3 = 27$. However, since an array of size 3 has $3! = 6$ distinct permutations, and 27 is not divisible by 6, it is not possible that each of the 6 permutations of A has the same probability of happening among the 27 possible outcomes of PERMUTE-WITH-ALL.

5.3-4 Professor Armstrong suggests the following procedure for generating a uniform random permutation:

```
Permute-By-Cyclic($A$)
  $n = A.length$
  let $B[1 \ldots n]$ be a new array
  $offset$ = Random($1, n$)
  for $i = 1$ to $n$ do
      $dest = i + offset$
      if dest $> n$ then
          $dest = dest - n$
      $B[dest] = A[i]$
```

Show that each element $A[i]$ has a $1/n$ probability of winding up in any particular position in B. Then show that Professor Armstrong is mistaken by showing that the resulting permutation is not uniformly random.

What Professor Armstrong's code does is a circular shift of all the elements to the right by i positions. Since each of the n possible shifts has the same probability of happening, each element has indeed a probability of $1/n$ of winding up in any particular position of the final array B. However, since this code has only n possible outcomes and A has $n!$ permutations, it can not produce a uniform random distribution over A. More precisely, the Professor Armstrong's code is not able to produce any permutation of A that is not a circular shift of A.

5.3-5 ($\star$) Prove that in the array P in procedure Permute-By-Sorting, the probability that all elements are unique is at least $1 - 1/n$.

Let X_i be an indicador random variable for the event that the ith priority is not unique. Since the subarray $P[1, \ldots, i-1]$ has at most $i-1$ distinct numbers, we have $\Pr\{X_i\} = \mathrm{E}[X_i] \leq (i-1)/n^3$. Let X be a random variable for the event that at least one priority is not unique. Then

$$X = (X_1 \cup X_2 \cup \cdots X_n) = X_1 + X_2 + \cdots + X_n,$$

which implies

$$\begin{aligned} \mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{n} X_i\right] \\ &= \sum_{i=1}^{n} \mathrm{E}[X_i] \\ &\leq \sum_{i=1}^{n} \frac{i-1}{n^3} \\ &= \frac{1}{n^3} \sum_{i=0}^{n-1} i \\ &= \frac{1}{n^3} \frac{(n-1) \cdot n}{2} \\ &= \frac{n-1}{2n^2} \\ &\leq \frac{1}{n}. \end{aligned}$$

Thus, the probability that all elements are unique is

$$\mathrm{E}[\overline{X}] = 1 - \mathrm{E}[X] \geq 1 - \frac{1}{n}.$$

5.3-6 Explain how to implement the algorithm PERMUTE-BY-SORTING to handle the case in which two or more priorities are identical. That is, your algorithm should produce a uniform random permutation, even if two or more priorities are identical.

The pseudocode is stated below.

```
Permute-By-Sorting-Unique($A$)
1   $n = A.length$
2   let $P[1 \dots n]$ be a new array
3   repeat
4       for $i = 1$ to $n$ do
5           $P[i] =$ Random$(1, n^3)$
6       let $Q$ be a copy of $P$
7       sort $Q$
8       $unique =$ True
9       for $i = 2$ to $n$ do
10          if $Q[i] == Q[i-1]$ then
11              $unique =$ False
12              break
    until $unique$
13  sort $A$, using $P$ as sort keys
```

Before sorting A using P as sort keys, the above algorithm verifies if P has unique priorities. If the priorities are not unique, P is generated again until it has unique priorities. Since the probability that a random P is unique is at least $1 - 1/n$, the expected number of iterations of the repeat loop of lines 3-12 is less than 2.

5.3-7 Suppose we want to create a ***random sample*** of the set $\{1, 2, 3, \ldots, n\}$, that is, an m-element subset S, where $0 \le m \le n$, such that each m-subset is equally likely to be created. One way would be to set $A[i] = i$ for $i = 1, 2, 3, \ldots, n$, call RANDOMIZED-IN-PLACE(A), and then take just the first m array elements. This method would make n calls to the RANDOM procedure. If n is mucn larger than m, we can create a random sample with fewer calls to RANDOM. Show that the following recursive procedure returns a random m-subset S of $\{1, 2, 3, \ldots, n\}$, in which each m-subset is equally likely, while making only m calls to RANDOM:

```
Random-Sample(m, n)
1  if $m == 0$ then
2      return $\emptyset$
3  else
4      $S$ = Random-Sample($m-1, n-1$)
5      $i$ = Random($1, n$)
6      if $i \in S$ then
7          $S = S \cup \{n\}$
8      else
9          $S = S \cup \{i\}$
10     return $S$
```

The recursion has $m+1$ levels. Let R_k, for $k = 0, 1, \ldots, m$, denote the recursion at depth k, in which an k-subset is returned (R_0 returns the empty set; R_m returns the final m-subset). After R_k, S will consist of k elements from the set $\{1, 2, \ldots, n-(m-k)\}$. There are $\binom{n-(m-k)}{k}$ ways to choose k elements from an $(n-(m-k))$-set. Thus, to S be a random sample, we wish to show that, in each recursion level k, this particular k-subset is selected with probability $1/\binom{n-(m-k)}{k}$.

For the base case of the recursion, which occurs when $k = 0$, there are $\binom{n-m}{0} = 1$ distincts 0-subsets and the algorithm returns the empty set with probability $1 = 1/\binom{n-m}{0}$. Now assume R_{k-1} returns an random $(k-1)$-sample. There are two ways to add the kth element to S on R_k:

- The element $n-(m-k)$ is added. This occurs when line 5 either selects the element $n-(m-k)$ or an element e such that $e \in R_{k-1}$. This probability is

$$\underbrace{\frac{1}{n-(m-k)}}_{(n-(m-k))\text{ is selected}} + \underbrace{\frac{k-1}{n-(m-k)}}_{e \in R_{k-1}\text{ is selected}} = \frac{k}{n-(m-k)}.$$

  Thus, R_k produces a particular k-sample with the element $n-(m-k)$ with probability

$$\begin{aligned}\frac{k}{n-(m-k)} \cdot \frac{1}{\binom{n-(m-k)-1}{k-1}} &= \frac{k}{n-(m-k)} \cdot \left(\frac{(n-(m-k)-1)!}{(k-1)! \cdot (n-(m-k)-1-(k-1))}\right)^{-1} \\ &= \left(\frac{(n-(m-k))!}{k! \cdot (n-(m-k)-k)}\right)^{-1} \\ &= \frac{1}{\binom{n-(m-k)}{k}}.\end{aligned}$$

- An element $j < n-(m-k)$ is added. The probability of line 5 selecting such element is

$$\frac{n-(m-k)-k}{n-(m-k)} = \frac{n-m}{n-(m-k)}.$$

  Thus, R_k produces a particular k-sample with the element j with probability

$$\begin{aligned}\frac{n-m}{n-(m-k)} \cdot \frac{1}{\binom{n-(m-k)-1}{k}} &= \frac{n-m}{n-(m-k)} \cdot \left(\frac{(n-(m-k)-1)!}{k! \cdot (n-(m-k)-1-k)}\right)^{-1} \\ &= \left(\frac{(n-(m-k))!}{k! \cdot (n-(m-k)-k)}\right)^{-1} \\ &= \frac{1}{\binom{n-(m-k)}{k}}.\end{aligned}$$

Since each recursion level R_k such that $k > 0$ makes exactly one call to RANDOM, there are m such calls. Also, among the $\binom{n}{m}$ ways of choosing m elements from an n-set, RANDOM-SAMPLE returns each of them with probability

$$\frac{1}{\binom{n-(m-m)}{m}} = \frac{1}{\binom{n}{m}}.$$

Problems

5-1 ***Probabilistic counting***
With a b-bit counter, we can ordinarily only count up to $2^b - 1$. With R. Morri's ***probabilistic counting***, we can count up to a much larger value at the expense of some loss of precision.

We let a counter value of i represent a count of n_i for $i = 0, 1, \ldots, 2^b - 1$, where the n_i form an increasing sequence of nonnegative values. We assume that the initial value of the counter is 0, representing a count of $n_0 = 0$. The INCREMENT operation works on a counter containing the value i in a probabilistic manner. If $i = 2^b - 1$, then the operation reports an overflow error. Otherwise, the INCREMENT operation increases the counter by 1 with probability $1/(n_{i+1} - n_i)$.

If we select $n_i = i$ for all $i \geq 0$, then the counter is an ordinary one. More interesting situations arise if we select, say, $n_i = 2^{i-1}$ for $i > 0$ or $n_i = F_i$ (the ith Fibonacci number – see Section 3.2).

For this problem, assume n_{2^b-1} is large enough that the probability of an overflow error is negligible.

a. Show that the expected value represented by the counter after n INCREMENT operations have been performed is exactly n.

b. The analysis of the variance of the count represented by the counter depends on the sequence of the n_i. Let us consider a simple case: $n_i = 100i$ for all $i \geq 0$. Estimate the variance in the value represented by the register after n INCREMENT operations have been performed.

(a) Let X_i denote a random variable for the expected *increment* of the count represented by a counter of value i after *one* INCREMENT operation. We have

$$\mathrm{E}[X_i] = 0 \cdot \left(1 - \frac{1}{n_{i+1} - n_i}\right) + (n_{i+1} - n_i) \cdot \frac{1}{n_{i+1} - n_i} = 1,$$

which shows that, independently from the current state of the *counter*, the expected *increment* of the *count* after each INCREMENT operation is always 1. Thus, after n INCREMENT operations, the expected *count* is:

$$\sum_{i=1}^{n} \mathrm{E}[X_0] = \sum_{i=1}^{n} 1 = n,$$

(b) We have

$$\begin{aligned}\mathrm{Var}[X_i] &= \mathrm{E}[{X_i}^2] - \mathrm{E}^2[X_i] \\ &= \left(0^2 \cdot \left(1 - \frac{1}{100}\right) + 100^2 \cdot \frac{1}{100}\right) - 1 \\ &= 99,\end{aligned}$$

which shows that the estimated variance after each INCREMENT operation does not depend on the current state of the *counter*. Thus, after n INCREMENT operations, the estimated variance is

$$\sum_{i=1}^{n} \mathrm{Var}[X_0] = \sum_{i=1}^{n} 99 = 99n.$$

5-2 ***Searching an unsorted array***

This problem examines three algorithms for searching for a value x in an unsorted array A consisting of n elements.

Consider the following randomized strategy: pick a random index i into A. If $A[i] = x$, then we terminate; otherwise, we continue the search by picking a new random index into A. We continue picking random indices into A until we find an index j such that $A[j] = x$ or until we have checked every element of A. Note that we pick from the whole set of indices each time, so that we may examine a given element more than once.

a. Write pseudocode for a procedure RANDOM-SEARCH to implement the strategy above. Be sure that your algorithm terminates when all indices into A have been picked.

b. Suppose that there is exactly one index i such that $A[i] = x$. What is the expected number of indices into A that we must pick before we find x and RANDOM-SEARCH terminates?

c. Generalizing your solution to part (b), suppose that there are $k \geq 1$ indices i such that $A[i] = x$. What is the expected number of indices into A that we must pick before we find x and RANDOM-SEARCH terminates? Your answer should be a function of n and k.

d. Suppose that there are no indices i such that $A[i] = x$. What is the expected number of indices into A that we must pick before we have checked all elements of A and RANDOM-SEARCH terminates?

Now consider a deterministic linear serach algorithm, which we refer to as DETERMINISTIC-SEARCH. Specifically, the algorithm searches A for x in order, considering $A[1], A[2], A[3], \ldots, A[n]$ until either it finds $A[i] = x$ or it reaches the end of the array. Assume that all possible permutations of the input array are equally likely.

e. Suppose that there is exactly one index i such that $A[i] = x$. What is the average-case running time of DETERMINISTIC-SEARCH? What is the worst-case running time of DETERMINISTIC-SEARCH?

f. Generalizing your solution to part (e), suppose that there are $k \geq 1$ indices i such that $A[i] = x$. What is the average-case running time of DETERMINISTIC-SEARCH? What is the worst-case running time of DETERMINISTIC-SEARCH? Your answer should be a function of n and k.

g. Suppose that there are no indices i such that $A[i] = x$. What is the average-case running time of DETERMINISTIC-SEARCH? What is the worst-case running time of DETERMINISTIC-SEARCH?

Finally, consider a randomized algorithm SCRAMBLE-SEARCH that works by first randomly permuting the input array and then running the deterministic linear search given above on the resulting permuted array.

h. Letting k be the number of indices i such that $A[i] = x$, give the worst-case and expected running times of SCRAMBLE-SEARCH for the cases in which $k = 0$ and $k = 1$. Generalize your solution to handle the case in which $k \geq 1$.

i. Which of the three searching algorithms would you use? Explain your answer.

(a) The pseudocode is stated below.

```
Random-Search(A, x)
    $I = \emptyset$
    $n = A.length$
    $index = -1$
    while $|I| < n$ do
        $i = \text{Random}(1, n)$
        $I = I \cup \{i\}$
        if $A[i] == x$ then
            $index = i$
            break
    return $index$
```

(b) This can be viewed as a sequence of Bernoulli trials, each with a probability $p = 1/n$ of success. Let X be a random variable for the number of trials needed to pick i such that $A[i] = x$. From Equation (C.32), we have

$$\mathrm{E}[X] = \frac{1}{p} = n.$$

(c) This can also be viewed as a sequence of Bernoulli trials, but with a probability $p = k/n$ of success. Thus, we have

$$\mathrm{E}[X] = \frac{1}{p} = \frac{n}{k}.$$

(d) Let I be the set of indexes that was already checked. Let X_i be a random variable for the number of trials needed to pick an index i, for $i = 1, 2, \ldots, n$, such that $i \notin I$ and $|I| = i - 1$. This can be viewed as a sequence of Bernoulli trials. Thus, we have

$$p = \frac{n - |I|}{n} = \frac{n - i + 1}{n},$$

and

$$\mathrm{E}[X_i] = \frac{1}{p} = \frac{n}{n - i + 1}.$$

Now let X be a random variable for the number of trials to pick all elements of A. We have

$$\begin{aligned}
\mathrm{E}[X] = \mathrm{E}\left[\sum_{i=1}^{n} X_i\right] &= \sum_{i=1}^{n} \mathrm{E}[X_i] \\
&= \sum_{i=1}^{n} \frac{n}{n - i + 1} \\
&= n \sum_{i=1}^{n} \frac{1}{n - i + 1} \\
&= n \sum_{i=0}^{n-1} \frac{1}{n - i} \\
&= n \sum_{i=1}^{n} \frac{1}{i} \qquad \text{(nth harmonic number)} \\
&= n(\ln n + O(1)).
\end{aligned}$$

(e) Lets first consider the average case. Among the $n - 1$ elements that is not x, $(n - 1)/2$ of them are expected to be before the element x on the array. Thus, the expected running time of the algorithm is

$$\frac{n - 1}{2} + 1 = \frac{n + 1}{2}.$$

The worst-case occur when the number of elements before x is $n - 1$. In this case, the algorithm will make n checks.

(f) Let I be the set of indexes such that $i \in I \to A[i] = x$. For each element e such that $e \neq x$, there are $k + 1$ possibilities to position e with respect to I (before all elements of I, after one element of I, but before the remaining $k - 1$ elements of I, and so on). Each of these positions is equally likely. Therefore, among the $n - k$ elements that is not x, $(n - k) \cdot 1/(k + 1) = (n - k)/(k + 1)$ are expected to be before all the elements of I. Thus, the expected running time of the algorithm is

$$\frac{n - k}{k + 1} + 1 = \frac{(n - k) + (k + 1)}{k + 1} = \frac{n + 1}{k + 1}.$$

The worst-case occurs when the number of elements before the first x is $n - k$. In this case, the algorithm will make $n - k + 1$ checks.

(g) In every case, the algorithm will check all elements of A. Thus, there will be n checks.

(h) Suppose the algorithm uses Randomize-In-Place to randomize the input array. Independently from the value of k, the algorithm will take n on this operation. Thus, lets focus on the number of checks for each case. When $k = 0$, the algorithm will make exactly n checks in every case. Thus, it the expected running time is $n + n = 2n$. When $k = 1$, the behaviour of the algorithm is similar to the one of item (e). Thus, the expected running time is $n + (n+1)/2 = (3n+1)/2$. As for the worst-case, note that this notation refers to the distribution of inputs. Since for every input the expected running time is the same, the worst-case (over the inputs) is $n + (n + 1)/2 = (3n + 1)/2$. Similarly, for a given k and from item (f), both the expected running time and the worst-case is $n + (n + 1)/(k + 1)$.

(i) Deterministic-Search is better in all cases.