Exercises in Section 8.1

1. What is the smallest possible depth of a leaf in a decision tree for a comparison sort?
   Solution We can think the decision tree as a construction of a total order (that is, every pair of elements is comparable) between n elements. The root represents the comparison of two elements, which we call the ordered elements. In each of the following nodes, either a new element is related with the ordered elements, or two new elements are ordered separately from the ordered elements and we need another comparison to relate them to the ordered elements. Thus, there are at least $n-1$ comparison needed to establish a total order between n elements.

2. Obtain asymptotically tight bounds on $\lg(n!)$ without using Stirling's approximation. Instead, evaluate the summation $\sum_{k=1}^{n} \lg k$ using techniques from Section A.2.
   Solution First, we notice that

$$\lg(n!) = \sum_{k=1}^{n} \lg k.$$

   We aim to show its asymptotically tight bound is $\Theta(n \lg n)$. The upper bound is trivial:

$$\sum_{k=1}^{n} \lg k \le \sum_{k=1}^{n} \lg n = n \lg n = O(n \lg n).$$

   For simplicity, we assume n is even. By considering only half of the summation, the lower bound is shown:

$$\sum_{k=1}^{n} \lg k = \sum_{k=n/2}^{n} \lg(n/2) = (n/2)\lg(n/2) = \Omega(n \lg n).$$

   Hence, $\sum_{k=1}^{n} \lg k = \Theta(n \lg n)$. □

3. Show that there is no comparison sort whose running time is linear for at least half of the $n!$ inputs of length n. What about a fraction of $1/n$ of the inputs of length n? What about a fraction $1/2^n$?
   Solution Suppose there is a comparison sort whose running time is linear for $kn!$ of the $n!$ inputs of length n. In the (binary) decision tree, there are at most 2^{cn} leaves of level cn. Therefore,

$$kn! \le 2^{cn}$$

for all n, and for a constant c. Or, equivalently,

$$kn! = O(2^n).$$

Consider the cases:

(i) $k = 1/2$. We have

$$kn! = n!/2 = \Theta(n!) = \omega(2^n).$$

(ii) $k = 1/n$. We have

$$kn! = (n-1)! = \Theta(n!) = \omega(2^n).$$

(iii) $k = 1/2^n$. We have

$$\lim_{n\to\infty} \frac{kn!}{2^n} = \lim_{n\to\infty} \frac{n!}{4^n} = \infty.$$

Thus, $kn! = \omega(2^n)$.

In all cases, we have $kn! = \omega(2^n)$. Hence, no comparison sort has linear running time for $1/2$, $1/n$, or $1/2^n$ of the $n!$ inputs of length n.

4. You are given an n-element input sequence, and you know in advance that it is partly sorted in the following sense. Each element initially in position i such that $i \bmod 4 = 0$ is either already in its correct position, or it is one place away from its correct position. For example, you know that after sorting, the element initially in position 12 belongs in position 11, 12, or 13. You have no advance information about the other elements, in positions i where $i \bmod 4 \neq 0$. Show that an $\Omega(n \lg n)$ lower bound on comparison based sorting still holds in this case.

Solution Let us count the number of permutations that satisfies the condition mentioned. There are approximately $n/4$ elements whose positions is divided by 4, and therefore each of them has three possible positions after sorts. In total, there are approximately $3^{n/4}$ cases. There is no other restrictions, and other elements have approximately $(3n/4)!$ ways to permutate. Thus, the total number of permutations is approximately $3^{n/4}(3n/4)!$. Now, since each permutation must appear in at least one leaf, and the total number of leaves is at most 2^h, we have

$$3^{n/4}(3n/4)! \leq 2^h.$$

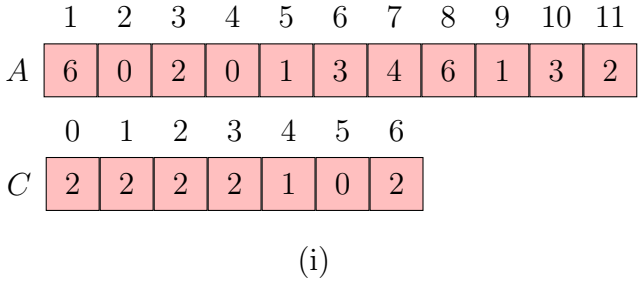
Taking logarithms on both sides, we have

$$\begin{aligned} h &\geq \frac{\lg 3}{4} n + \Theta((3n/4) \lg (3n/4)) \\ &= \Theta(n) + \Theta(n \lg n) \\ &= \Omega(n \lg n). \end{aligned}$$

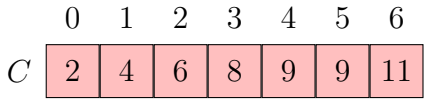
Exercises in Section 8.2

1. Using Figure 8.2 as a model, illustrate the operation of COUNTING-SORT on the array $A = \langle 6, 0, 2, 0, 1, 3, 4, 6, 1, 3, 2 \rangle$.

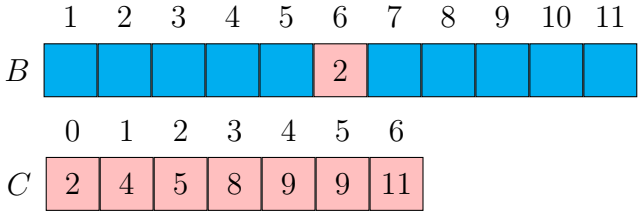
Solution



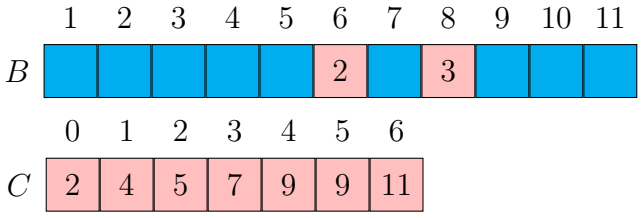
(i)



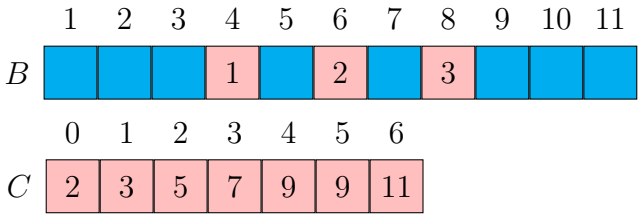
(ii)



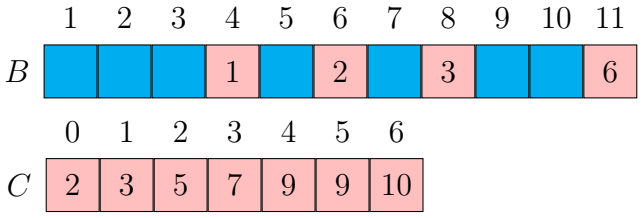
(iii)



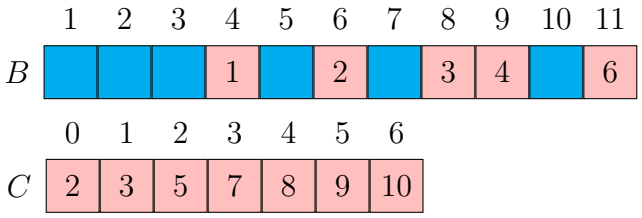
(iv)



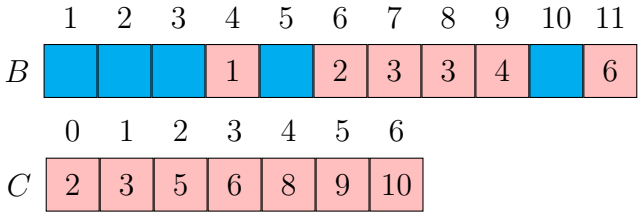
(v)



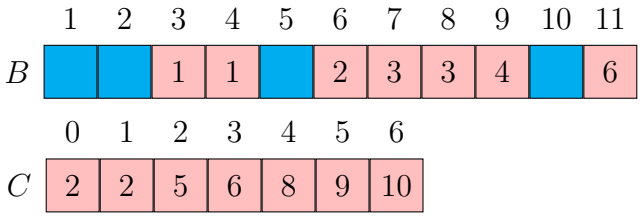
(vi)



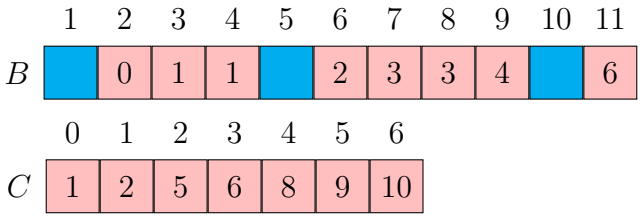
(vii)



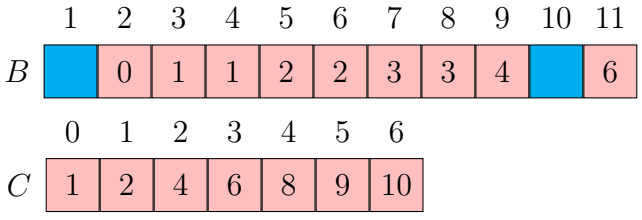
(viii)



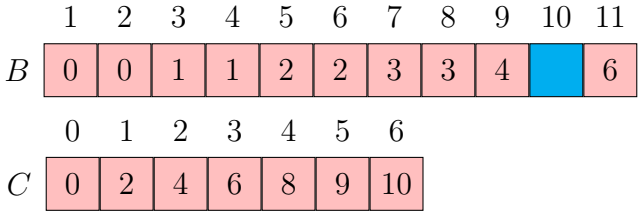
(ix)



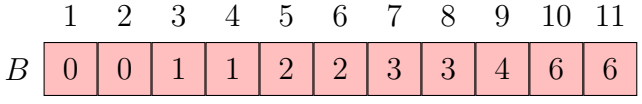
(x)



(xi)



(xii)



(xiii)

2. Prove that COUNTING-SORT is stable.

Solution

It suffices to show that for any two equal elements, say $A[i] = A[j] = x$, where $i < j$, the index of $A[i]$ is less than $A[j]$. Note that the loop in line 11-13 applies downward iterations, that is, $A[j]$ is iterated before $A[i]$. Line 13 then guarantees $C[x]$, the index, is lower for $A[i]$ than it was for $A[j]$. □

3. Suppose that we were to rewrite the **for** loop header in line 11 of the COUNTING-SORT as

for $j = 1$ **to** n

Show that the algorithm still works property, but that it is not stable. Then rewrite the pseudocode for counting sort so that elements with the same value are written into the output array in order of increasing index and the algorithm is stable.

Solution

With forward iteration, the sorting algorithm still works, because the order of elements in the original array A does not affect the correctness of the output (you can even iterate A in any weird order). However, for two elements $A[i]$ and $A[j]$ with the same value, where $i < j$, $A[i]$ will appear later than $A[j]$ with similar reason as in the previous question. So, the new algorithm is not stable. Below shows an updated algorithm that is stable:

Algorithm 1 FORWARD-COUNTING-SORT(A, n, k)

```
1: let B[1 : n] and C[0 : k] be new arrays
2: for i = 0 to k do
3:     C[i] = 0
4: end for
5: for j = 1 to n do
6:     C[A[j]] = C[A[j]] + 1
7: end for
8: for i = 1 to k do
9:     C[i] = C[i] + C[i − 1]
10: end for
11: for i = k downto 1 do
12:     C[i] = C[i − 1] + 1
13: end for
14: C[0] = 1
15: for j = 1 to n do
16:     B[C[A[j]]] = A[j]
17:     C[A[j]] = C[A[j]] + 1
18: end for
19: return B
```

4. Prove the following loop invariant for COUNTING-SORT: At the start of each iteration of the **for** loop of lines 11-13, the last element in A with value i that has not yet been copied into B belongs in $B[C[i]]$.

Solution

- Initialization: Before the first iteration, $C[i]$ is the number of elements with value less than or equal to i. Therefore, the last element with value i should be located to $B[C[i]]$, the last slot for value i.
- Maintenance: Suppose we are at index j and $A[j] = i$, and $A[j]$ belongs to $B[C[i]]$ in this iteration. In line 13, $C[i]$ is reduced by 1, therefore, $B[C[i]]$ should exactly contain the next element with value i. □

5. Suppose that the array being sorted contains only integers in the range 0 to k and that there are no satellite data to move with those keys. Modify counting sort to use just the arrays A and C, putting the sorted result back into array A instead of into a new array B.

Solution

Algorithm 2 MODIFIED-COUNTING-SORT(A, n, k)

```
1: let C[0 : k] be a new array
2: for i = 0 to k do
3:     C[i] = 0
4: end for
5: for j = 1 to n do
6:     C[A[j]] = C[A[j]] + 1
7: end for
8: index = 1
9: for i = 0 to k do
10:    for j = 1 to C[i] do
11:        A[index] = i
12:        index = index + 1
13:    end for
14: end for
15: return A
```

6. Describe an algorithm that, given n integers in the range 0 to k, preprocesses its input and then answers any query about how many of the n integers fall into a range $[a : b]$ in $O(1)$ time. Your algorithm should use $\Theta(n + k)$ preprocessing time.

Solution

Given an interval $[a : b]$, the key idea is to find the number of integers that are larger than or equal to a, and the number of integers that are smaller than or equal to b. Basic set theory then tells us the answer equals the sum of these two numbers minus n. The preprocessing algorithm:

Algorithm 3 PREPROCESS-INTEGERS(A, n, k)

```
let C[0 : k] and D[0 : k] be new arrays
for i = 0 to k do
    C[i] = 0
    D[i] = 0
end for
for j = 1 to n do
    C[A[j]] = C[A[j]] + 1
    D[A[j]] = D[A[j]] + 1
end for
for i = 1 to k do
    C[i] = C[i] + C[i − 1]
end for
for i = k − 1 downto 0 do
    D[i] = D[i] + D[i + 1]
end for
return C, D
```

Once we have the arrays C and D, the number of integers falling in $[a : b]$ is equal to

$$C[a] + D[b] - n,$$

finishing in $O(1)$ time.

7. Counting sort can also work efficiently if the input values have fractional parts, but the number of digits in the fractional part is small. Suppose that you are given n number in the range 0 to k, each with at most d decimal (base 10) digits to the right of the decimal point. Modify counting sort to run in $\Theta(n + 10^d k)$ time.

Solution

The key idea is to multiply each decimal number by 10^d, therefore we are dealing with n integers in the range 0 to $10^d k$. For this reason, we will call the original counting sort algorithm in the modified algorithm:

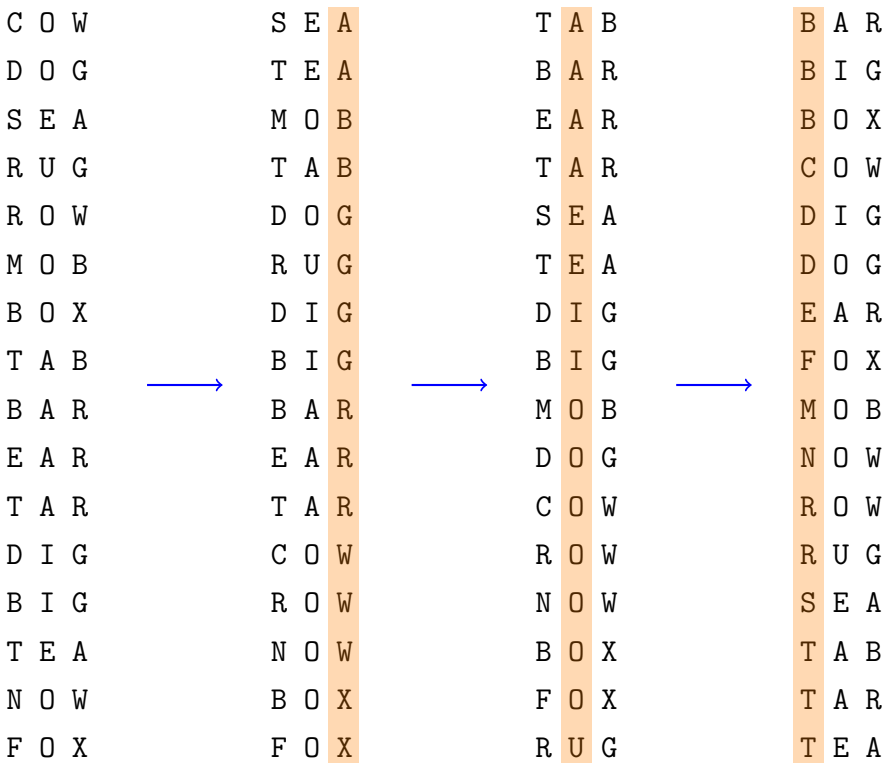
Algorithm 4 MODIFIED-COUNTING-SORT(A, n, k, d)

1: let $B[1:n]$ be a new array
2: **for** $i = 0$ to n **do**
3: $\quad B[i] = 10^d * A[i]$
4: **end for**
5: **return** COUNTING-SORT$(B, n, 10^d * k)$

Exercises in Section 8.3

1. Using Figure 8.3 as a model, illustrate the operation of RADIX-SORT on the following list of English words: COW, DOG, SEA, RUG, ROW, MOB, BOX, TAB, BAR, EAR, TAR, DIG, BIG, TEA, NOW, FOX.

Solution



2. Which of the following sorting algorithms are stable: insertion sort, merge sort, heapsort, and quicksort? Give a simple scheme that makes any comparison sort stable. How much additional time and space does your scheme entail?

Solution

- Stable: insertion sort, merge sort.
- Unstable: heapsort, quicksort.

To make any comparison sort stable, we extend A into a subset $A' \subseteq A \times \{0, 1, \ldots, n-1\}$, where for each index $i \in \{0, 1, \ldots, n-1\}$, if $x = A[i]$, then $A'[i] = (x, i)$. Then, we define the order $(x, i) < (y, j)$ if and only if $x \leq y$ and $i < j$. Therefore, any comparison sort is stable when it is applied on A' instead of A.

The additional time depends on the choice of sorting method. It should be at least $\Theta(n)$, which involves the construction of A', and at most the time complexity of the sorting method chosen, because each comparison still take $\Theta(1)$ time.

The space is (almost) doubled due to the use of pairs.

3. Use induction to prove that radix sort works. Where does your proof need the assumption that the intermediate sort is stable?

Solution

- If $d = 1$, radix sort is equivalent to counting sort (or other stable sort).
- Assume radix sort is correct and stable for $d - 1$, that is, radix sort is correct and stable in sorting $(d-1)$-digit numbers. Consider d-digit numbers (denote x_l as the l-th digit of x):
  - Correctness: for any two numbers x and y, if $x_1 \neq y_1$, x_1 and y_1 are placed correctly by counting sort (or other stable sort). If $x_1 = y_1$, the stability of counting sort (or other stable sort) and the inductive assumption yields the correctness.
  - Stability: if $x = y$, the inductive assumption guarantees unchanged relative position in the first $d - 1$ iterations, and the stability of counting sort (or other stable sort) guarantees unchanged relative positions in the d-th iteration.

4. Suppose that COUNTING-SORT is used as the stable sort within RADIX-SORT. If RADIX-SORT calls COUNTING-SORT d times, then since each call of COUNTING-SORT makes two passes over the data (lines 4-5 and 11-13), altogether $2d$ passes over the data occur. Describe how to reduce the total number of passes to $d + 1$.

Solution

While placing the elements (lines 11-13) based on the l-th digit, we can simultaneously count (lines 4-5) the $(l-1)$-th digit. Therefore, the number of passes is reduced to $d + 1$.

5. Show how to sort n integers in the range 0 to $n^3 - 1$ in $O(n)$ time.

Solution

We can view each element in range $[0 : n^3 - 1]$ as a 3-digit number, where each digit ranges from 0 to $n - 1$ (in other words, turn the numbers to base n). Therefore, radix sort costs $\Theta(3(n + n)) = \Theta(n)$ time.

6. In the first card-sorting algorithm in this section, which sorts on the most significant digit first, exactly how many sorting passes are needed to sort d-digit decimal numbers in the worst case? How many piles of cards does an operator need to keep track of in the worst case?

Solution

We will sort n d-digit numbers, where each digit ranges from 0 to k, using the indicated algorithm. In the worst case, $n = k^d$, and each pile splits into $k + 1$ piles of the same number of cards. The total number of sorting passes is

$$\sum_{i=0}^{d-1} (k+1)^i = \frac{(k+1)^d - 1}{k} = \Theta(k^d) = \Theta(n).$$

The number of piles is

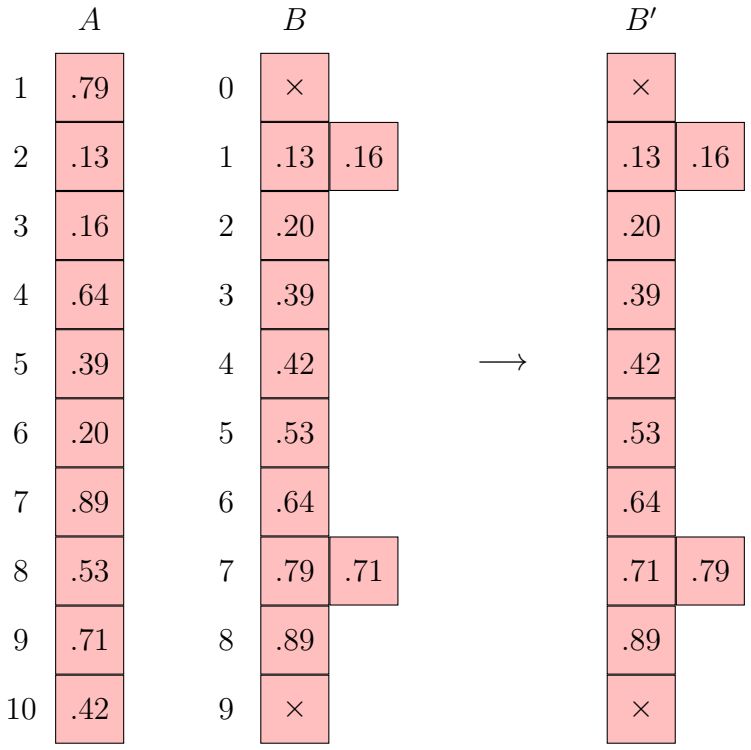
$$\sum_{i=0}^{d} (k+1)^i = \frac{(k+1)^{d+1} - 1}{k} = \Theta(k^{d+1}) = \Theta(nk).$$

Exercises in Section 8.4

1. Using Figure 8.4 as a model, illustrate the operation of BUCKET-SORT on the array $A = \langle .79, .13, .16, .64, .39, .20, .89, .53, .71, .42 \rangle$.

Solution

I don't like Figure 8.4.



Note that the arrow means insertion sort on each row of B.

2. Explain why the worst-case running time for bucket sort is $\Theta(n^2)$. What simple change to the algorithm preserves its linear average-case running time and makes its worst-case running time $O(n \lg n)$?

Solution

If, coincidentally, all n elements fall in the same interval, say $B[0]$. Then, insertion sort on $B[0]$ would cost $\Theta(n^2)$ time.

To reduce the worst-case running time to $O(n \lg n)$, we can simply replace insertion sort by other sorting methods that run in $O(n \lg n)$ (e.g. merge sort, heapsort, quicksort).

3. Let X be a random variable that is equal to the number of heads in two flips of a fair coin. What is $E[X^2]$? What is $E^2[X]$?

Solution

Note that X follows binomial distribution. More precisely, $X \sim B(2, \frac{1}{2})$. Thus,

$$E[X] = 2\left(\frac{1}{2}\right) = 1, \quad \text{and} \quad \text{Var}[X] = 2\left(\frac{1}{2}\right)\left(1 - \frac{1}{2}\right) = \frac{1}{2}.$$

Therefore,

$$E^2[X] = 1^2 = 1,$$

and

$$E[X^2] = \text{Var}[X] + E^2[X] = \frac{1}{2} + 1 = \frac{3}{2}.$$

4. An array A of size $n > 10$ is filled in the following way. For each element $A[i]$, choose two random variables x_i and y_i uniformly and independently from $[0, 1)$. Then set

$$A[i] = \frac{\lfloor 10x_i \rfloor}{10} + \frac{y_i}{n}.$$

Modify bucket sort so that it sorts the array A in $O(n)$ expected time.

Solution

Modify line 5 from

$$\text{insert } A[i] \text{ into list } B[\lfloor n \cdot A[i] \rfloor]$$

to

$$\text{insert } A[i] \text{ into list } B[\lfloor \frac{n^2}{10} \cdot (10 \cdot A[i] - \lfloor 10 \cdot A[i] \rfloor) \rfloor],$$

where $10 \cdot A[i] - \lfloor 10 \cdot A[i] \rfloor$ is the decimal part of

$$10 \cdot A[i] = \lfloor 10x_i \rfloor + \frac{10}{n} y_i.$$

It is easy to see the decimal part is exactly $\frac{10}{n} y_i$. Finally,

$$\lfloor \frac{n^2}{10} \cdot \frac{10}{n} y_i \rfloor = \lfloor n \cdot y_i \rfloor.$$

Since y_i varies uniformly in $[0, 1)$, it aligns exactly with classic bucket sort.

5. You are given n points in the unit disk, $p_i = (x_i, y_i)$, such that $0 < x_i^2 + y_i^2 \leq 1$ for $i = 1, 2, \ldots, n$. Suppose that the points are uniformly distributed, that is, the probability of finding a point in any region of the disk is proportional to the area of that region. Design an algorithm with an average-case running time of $\Theta(n)$ to sort the n points by the distances $d_i = \sqrt{x_i^2 + y_i^2}$ from the origin. (*Hint:* Design the bucket sizes in BUCKET-SORT to reflect the uniform distribution of the points in the unit disk.)

Solution

Let $B[0]$ contain p_i where $0 < d_1 \leq \frac{1}{\sqrt{n}}$, $B[1]$ contain p_i where $\frac{1}{\sqrt{n}} < d_i \leq \frac{\sqrt{2}}{\sqrt{n}}$, $\ldots$, $B[n-1]$ contain p_i where $\frac{\sqrt{n-1}}{\sqrt{n}} < d_i \leq 1$. It is easy to check the areas of these regions are the same, yielding equal probability.

6. A ***probability distribution function*** $P(x)$ for a random variable X is defined by $P(x) = \Pr\{X \leq x\}$. Suppose that you draw a list of n random variables $X_1, X_2, \ldots, X_n$ from a continuous probability distribution function P that is computable in $O(1)$ time (given y you can find x such that $P(x) = y$ in $O(1)$ time). Given an algorithm that sorts these numbers in linear average-case time.

Solution

For each $i = 0, 1, \ldots, n-1$, let $y_i = \frac{i}{n}$, and find the corresponding x_i in $O(1)$ time. The total time cost for this step is $O(n)$ (in addition, define $x_n = 1$). Then, apply bucket sort with buckets $\{[x_i, x_{i+1})\}$, again in $O(n)$ time.