UM-SJTU Joint Institute
Introduction to Algorithms
VE477 Lab6 Report

Name: Taoyue Xia, ID:518370910087

Date: 2021/11/25

1. Random Search

a. Implementation in OCaml

```
let rec random_search k array h length num =
  if (Hashtbl.length h)  == length then
    (-1, num)
  else
    let i = Random.int length in
    let elem = List.nth array i in
    if elem == k then
      (i, num)
    else
      if Hashtbl.mem h i then
        random_search k array h length (num + 1)
      else
        begin
        Hashtbl.add h i true;
        random_search k array h length (num + 1)
        end
;;

let k = read_int();;
let sequence = read_line();;
let s = List.map int_of_string (Str.split (Str.regexp " ") sequence);;
let length = List.length s;;
let hash = Hashtbl.create length;;
Random.self_init ();;
let index, num = random_search k s hash length 1;;
if index == -1 then
  begin
  print_string "No such element in the array.\n";
  print_string "Total search time is: ";
  print_int num;
  print_string ".\n"
  end
else
  begin
  print_string "The index is ";
  print_int index;
  print_string ".\n";
  print_string "Total search time is: ";
  print_int num;
  print_string ".\n"
  end
```

It takes the manual input, and will return the index or `Not found`.

Sample output

```
(*Input*)
4
1 2 3 4 5 6 7 8 9 10
(*Output*)
The index is 3.
Total search time is: 1.
```

b. Determine average number of indices picked

Add the following three functions to help execute a loop of 100 times to get the average time. Also, the list will be generated randomly instead of manual input, with 1000 integers ranging from 0 to 1000.

```
let get_two (_, a) = a;;

let rec loop result list k h n =
  if n == 100 then result
  else
    let tuple = random_search k list h (List.length list) 1 in
    let num = get_two tuple in
    begin
      Hashtbl.clear h;
      loop (num :: result) list k h n
    end
;;

let rec generate_list result num =
  if num == 1000 then result
  else
    let x = (Random.int 1000) in
    generate_list (x :: result) (num + 1)
;;
```

i. k not in list

By running the code, we get the following outputs:

```
The average number of indices is: 7513.
The average number of indices is: 7540.
The average number of indices is: 7580.
```

So we can see that the average number of indices are about 7500 - 7600.

ii. one k in list

Sample outputs:

```
The average number of indices is: 505.
The average number of indices is: 486.
The average number of indices is: 974.
The average number of indices is: 1004.
The average number of indices is: 325.
```

We can see that the average number of indices varies, but most of the times around 500.

iii. More than one k in list

For two k in list, the output is like

```
The average number of indices is: 465.
The average number of indices is: 248.
The average number of indices is: 507.
The average number of indices is: 252.
The average number of indices is: 491.
The average number of indices is: 328.
The average number of indices is: 496.
```

For three k in list, the output is like:

```
The average number of indices is: 340.
The average number of indices is: 258.
The average number of indices is: 325.
The average number of indices is: 329.
The average number of indices is: 201.
The average number of indices is: 251.
The average number of indices is: 348.
The average number of indices is: 243.
```

From the above two cases, we can find that, when the number of elements which equal to k increases, the average number of indices decreases.

By approximation, if there exists n elements which equal to k, the average number would be about $\frac{1000}{n+2}$.

2. Linear Search

a. Implementation in OCaml

```
let rec linear_search k array i =
  match array with
  | [] -> -1
  | head :: tail ->
    if head == k then
      i
    else
      linear_search k tail (i+1)
;;

```

```
let k = read_int();;
let sequence = read_line();;
let s = List.map int_of_string (Str.split (Str.regexp " ") sequence);;
let length = List.length s;;
let index = linear_search k s 1;;
if index == -1 then
  print_string "No such element in the array.\n"
else
  begin
  print_string "The index is ";
  print_int index;
  print_string ".\n"
  end
```

Sample Output

```
(*Input*)
4
1 2 3 4 5 6 7 8 9 10
(*Output*)
The index is 3.
Total search time is: 4.
```

b. Average-case and Worst-case

Suppose that there are n elements in the array.

i. No k in list

- Average-case: n
- Worst-case: n

ii. One k in list

- Average-case: $n/2$
- Worst-case: n

iii. More than 1 k in list

Suppose there are i elements that are equal to k.

- Average-case:

  If the first element equals to k, it will happen with probability

$$\frac{\binom{n-1}{i-1}}{\binom{n}{i}}$$

If the second one is the first to be detected as k, then it will happen with probability:

$$\frac{\binom{n-2}{i-1}}{\binom{n-1}{i}}$$

Therefore, the average case can be expressed as:

$$\begin{aligned} avg &= \sum_{j=1}^{n-i} \frac{\binom{n-j}{i-1}}{\binom{n-j+1}{i}} \cdot j \\ &= \sum_{j=1}^{n-i} \frac{i}{n-j+1} \cdot j \end{aligned}$$

- Worst-case: $n - i$.

3. Scramble Search

a. Implementation in OCaml

```
let rec permutation list =
  let rec select rest n = function
  | [] -> raise Not_found
  | head :: tail ->
    if n == 0 then (head, rest @ tail)
    else select (head :: rest) (n - 1) tail
  in
  let rec select_rand l length =
    select [] (Random.int length) l
  in
  let rec loop result length list =
    if length == 0 then result
    else
      let elem, rest = select_rand list length in
      loop (elem :: result) (length - 1) rest
  in
  loop [] (List.length list) list
;;

let rec linear_search k array i =
  match array with
  | [] -> -1
  | head :: tail ->
    if head == k then
      i
    else
      linear_search k tail (i+1)
;;


let k = read_int();;
let sequence = read_line();;
```

```
let l = List.map int_of_string (Str.split (Str.regexp " ") sequence);;
Random.self_init ();;
let perm = permutation l;;
List.iter (fun x -> print_int x; print_string" ") perm;;
let index = linear_search k perm 1;;
if index == -1 then print_string "No such element in the array.\n"
else
  begin
    print_string "The index is ";
    print_int index;
    print_string ".\n"
  end
```

Sample Output

```
(*Input*)
4
1 2 3 4 5 6 7 8 9 10
(*Output*)
Permutation list: 9 10 5 3 1 8 2 7 6 4
The index is 10.
```

b. Average-case and Worst-case

Since the `ScrambleSearch` first permute the original array A randomly into A', and then use `LinearSearch` to get the result. The average-case and worst-case should be the same as `LinearSearch`, since randomness cannot be determined. Each permutation has equal probability to be performed.

i. No k in array

- Average-case: n
- Worst-case: n

ii. One k in array

- Average-case: $n/2$
- Worst-case: n

iii. More than 1 k in array

- Average-case:

$$\sum_{j=1}^{n-i} \frac{i}{n-j+1} \cdot j$$

- Worst-case: n

4. Best Complexity

Since the `random` generator follows a uniform distribution,

- In `RandomSearch`, each index is chosen with the same probability, so it takes more trials to get the final index.
- In `LinearSearch`, we just traverse the array to get the index, which is stable $\mathcal{O}(n)$ time complexity.
- In `ScrambleSearch`, we first create a permutation of the array, then perform `LinearSearch`. It can reduce some bad cases to good ones.

Therefore, `ScrambleSearch` seems to be the best strategy.

5. Test for three algorithms

i. Random Search

Code for test is shown below:

```
let rec random_search k array h length num =
  if (Hashtbl.length h)  == length then
    (-1, num)
  else
    let i = Random.int length in
    let elem = List.nth array i in
    if elem == k then
      (i, num)
    else
      if Hashtbl.mem h i then
        random_search k array h length (num + 1)
      else
        begin
        Hashtbl.add h i true;
        random_search k array h length (num + 1)
        end
;;

let get_two (_, a) = a;;

let rec loop result list k h n =
  if n == 1000 then result
  else
    let tuple = random_search k list h (List.length list) 1 in
    let num = get_two tuple in
    begin
      Hashtbl.clear h;
      loop (num :: result) list k h (n + 1)
    end
;;

let rec generate_list result num =
  if num == 10000 then result
  else
```

```
    let x = (Random.int 10000) in
    generate_list (x :: result) (num + 1)
;;


Random.self_init ();;
let k = Random.int 10000;;

let sequence = generate_list [] 0;;
let length = List.length sequence;;
let hash = Hashtbl.create length;;
let indices_list = loop [] sequence k hash 0;;
(* List.iter (fun x -> print_int x; print_string " ") indices_list;; *)
let avg = (List.fold_left (+) 0 indices_list) / 1000;;
print_string "The average number of indices is: ";;
print_int avg;;
print_string ".\n";;
```

Since for `RandomSearch`, it takes too many trials to get the index, we just test for the case of 1000 elements and 10000 elements.

For 1000 elements, we have the average:

```
The average number of indices is: 7497.
```

For 10000 elements, we have the average:

```
The average number of indices is: 97176.
```

Therefore, by approximation. If we want to test for 1000000 elements for 1000 times, the average would be about 7000000 to 10000000, which is about 10 times the size of the array.

ii. Linear Search

Code for test is shown below:

```
let rec linear_search k array i =
  match array with
  | [] -> 1000000
  | head :: tail ->
    if head == k then
      i
    else
      linear_search k tail (i+1)
;;


let rec generate_list result num =
  if num == 1000000 then result
  else
    let x = (Random.int 1000000) in
    generate_list (x :: result) (num + 1)
;;

```

```
let rec loop result num k =
  if num == 1000 then result
  else
    let s = generate_list [] 0 in
    let i = linear_search k s 1 in
    loop (i :: result) (num + 1) k
;;

Random.self_init ();;
let k = Random.int 1000000;;
let sequence = loop [] 0 k;;
let times = (List.fold_left (+) 0 sequence) / 1000;;
print_string "The number of search of 1000 times is: ";;
print_int times;;
print_string ".\n";;
```

Which generates an array with 1000000 elements randomly and run for 1000 times.

Finally, it gives the average seach times as:

```
The number of search of 1000 times is: 645338.
```

iii. Scramble Search

Code for test is shown below:

```
let safe_map f xs =
  let rec helper acc = function
    | [] -> List.rev acc
    | x :: xs ->
       helper (f x :: acc) xs
  in
  helper [] xs
;;

let shuffle list =
  let ll = safe_map (fun c -> (Random.bits (), c)) list in
  let sorted = List.sort compare ll in
  safe_map snd sorted
;;

let rec linear_search k array i =
  match array with
  | [] -> 1000000
  | head :: tail ->
    if head == k then
      i
    else
      linear_search k tail (i+1)
;;

(* let rec scramble_search k array i =
  let s = permutation array in
  linear_search k s i
;; *)
```

```
30
31 let rec generate_list result num =
32   if num == 1000000 then result
33   else
34     let x = (Random.int 1000000) in
35     generate_list (x :: result) (num + 1)
36 ;;
37
38 let rec loop result num k =
39   if num == 1000 then result
40   else
41     let s = generate_list [] 0 in
42     let perm = shuffle s in
43     let i = linear_search k perm 1 in
44     loop (i :: result) (num + 1) k
45 ;;
46
47 Random.self_init ();;
48 let k = Random.int 1000000;;
49 (* let sequence = generate_list [] 0;;
50 let s = shuffle sequence;;
51 List.iter (fun x -> print_int x; print_string " ") s;; *)
52
53 let indices = loop [] 0 k;;
54 let avg = (List.fold_left (+) 0 indices) / 1000;;
55 print_string "The number of search of 1000 times is: ";;
56 print_int avg;;
57 print_string ".\n";;
```

Which generates an array with 1000000 elements randomly and run for 1000 times.

Finally, it gives the average seach times as:

```
1 The number of search of 1000 times is: 628595.
```

a. Performance

According to the above statistics, we can see that `ScrambleSearch` has the best performance according to the number of search times.

But it takes quite long to construct the permutation. If we consider time consumption, then `LinearSearch` would be better.

b. Discussion

In item 4, I expect `ScrambleSearch` to be the best.

From the real test carried out in the previous part, it is true that `ScrambleSearch` is the best in terms of the number of search times.

However, in terms of time consumption, `LinearSearch` would be the best, since it doesn't need to perform a permutation which takes $\mathcal{O}(n)$ time complexity.