JOINT INSTITUTE

交大密西根学院

UM-SJTU Joint Institute

Introduction to Algorithms

VE477 Lab3 Report

Name: Taoyue Xia, ID:518370910087

Date: 2021/10/23

1. a. Sort and count

Code in OCaml is shown below:

```
let rec mergecount = function    (*merge and count*)
  | list, [] -> list, 0
  | [], list -> list, 0
  | x1::xs1, x2::xs2 ->
    if x1 <= x2 then
      match (mergecount (xs1, x2::xs2)) with
        | (list, count) -> x1 :: list, count
      else
        match (mergecount (x1::xs1, xs2)) with
        | (list, count) -> x2 :: list, count + (List.length xs1) + 1
;;
let rec split_at n tmp list =     (*split the list at specified position*)
  if n = 0 then List.rev tmp, list
  else match list with
  | [] -> List.rev tmp, []
  | head::tail -> split_at (n-1) (head::tmp) tail
;;

let split list = split_at ((List.length list + 1) / 2) [] list;;
(*split the list at middle*)

let get_ele1 (a, _) = a;;    (*get the first element of a two-element
tuple*)
let get_ele2 (_, a) = a;;    (*get the second element*)

let rec merge_sort = function    (*sort and count*)
    | [] -> [], 0
    | [_] as list -> list, 0
    | list ->
        let l1, l2 = (split list) in
          let (s1, s2) = merge_sort l1, merge_sort l2 in
            let (result, count) = mergecount (get_ele1 s1, get_ele1 s2) in
          result, get_ele2 (s1) + get_ele2 (s2) + count
;;

let rec print_list list = match list with    (*print function*)
  | [] -> print_string ""
  | head::[] -> print_int head
  | head::tail -> print_int head ; print_string ", " ; print_list tail
;;

let input = read_line();;
let ll = List.map int_of_string (Str.split (Str.regexp " ") input);;

let (list, count) = merge_sort ll;;
Printf.printf "%d\n" count;;
print_string "[";;
print_list list;;
print_string "]\n";;

```

OCaml features a recursion method, so using OCaml to write merge sort and count can save some time and the code will be more concise.

1. b. Gale-Shapley

Code in Ocaml is shown below:

```
type man =
  { m_index: int;
    mutable free: bool;
    women_rank: int list;
    has_proposed: (int, bool) Hashtbl.t
    (*Use hashtable to store proposed women*)
  }
;;
type woman =
  { w_index: int;
    men_rank: int list;
    mutable engaged: int option    (*can be either none or an index of man*)
  }
;;

(*The function to read inputs and convert to a list of tuples which contains an index and a rank list*)
let rec read_person i n ll =
  if i = n then List.rev ll
  else
    let s = read_line() in
      let l = List.map int_of_string (Str.split (Str.regexp " ") s) in
        match l with
        | [] -> List.rev ll
        | head::tail -> read_person (i+1) n ((i,l)::ll)
;;

(*returns true if m1 is of higher rank than m2 for w*)
let prefer w m1 m2 =
  let rec comp = function
  | [] -> invalid_arg "cannot find such index"
  | x::_ when x = m1 -> true
  | x::_ when x = m2 -> false
  | _::xs -> comp xs
  in
  comp w.men_rank
;;

(*initialize the struct on men and women*)
let build_structs men women n =
  List.map (fun (index, rank) -> {m_index = index; women_rank = rank; free = true; has_proposed = Hashtbl.create n}) men,
  List.map (fun (index, rank) -> {w_index = index; men_rank = rank; engaged = None}) women
;;

let gale_helper m_list w_list n =
```

```
45    (*Create two hashtables for men and women, key is the index, value is the
      corresponding struct*)
46    let men = Hashtbl.create n in
47    List.iter (fun m -> Hashtbl.add men m.m_index m) m_list;
48    let women = Hashtbl.create n in
49    List.iter (fun w -> Hashtbl.add women w.w_index w) w_list;
50    try
51      while true do
52        let m = List.find (fun m -> m.free) m_list in    (*Find the first free
      man from the list of men*)
53        let w_index = List.find (fun w -> not (Hashtbl.mem m.has_proposed w))
      m.women_rank in    (*Find the woman of highest rank who has not been
      proposed by m*)
54        Hashtbl.add m.has_proposed w_index true;    (*Add the woman to the
      proposed hashtable, key is index, value is en empty unit*)
55        let w = Hashtbl.find women w_index in     (*The hashtable can shorter
      the finding process to O(1)*)
56        match w.engaged with
57        | None -> w.engaged <- Some m.m_index; m.free <- false    (*If the
      woman is free, let m and w be a pair*)
58        | Some m'_index ->    (* some pair (m', w) already exists *)
59            if prefer w m.m_index m'_index then     (*If woman "w" prefer "m"
      to "m'"*)
60            begin
61              w.engaged <- Some m.m_index;
62              let m' = Hashtbl.find men m'_index in    (*Quick find in
      hashtable*)
63              m'.free <- true;
64              m.free <- false
65            end
66      done;
67    with Not_found -> ()
68  ;;
69
70  let gale men women n =
71    let m_list, w_list = build_structs men women n in
72    gale_helper m_list w_list n;
73    let some = function Some v -> v | None -> -1 in
74    List.stable_sort compare (List.map (fun w -> some w.engaged, w.w_index)
      w_list)
75  ;;
76
77  let get_ele1 (a, _) = a;;
78  let get_ele2 (_, a) = a;;
79  let rec print result =
80    match result with
81    | [] -> print_string ""
82    | head::[] -> print_string "["; print_int (get_ele1 head); print_string ",
      "; print_int (get_ele2 head); print_string "]"
83    | head::tail -> print_string "["; print_int (get_ele1 head); print_string
      ", "; print_int (get_ele2 head); print_string "], "; print tail
84  ;;
85
86  let () =
87    let n = read_int() in
88    let men= read_person 0 n [] in
89    let tmp = read_line() in
90    let women = read_person 0 n [] in
```

```
  let result = gale men women n in
  print_string tmp;
  print_string "[";
  print result;
  print_string "]\n"
;;
```

2. Knapsack Problem

a. Two strategies

(i). Smaller first

OCaml code:

```
let rec small_first list target result =
  match list with
  | [] -> []
  | head::tail ->
    if target = head then List.rev (head::result)
    else if target > head then small_first tail (target - head)
(head::result)
    else []
;;
```

Counter example:

input:

```
4
2 3 4
```

Output:

```
[]
```

But it should output `[4]`.

(ii). Larger first

OCaml code:

```
let rec large_first list target result =
  match list with
  | [] -> []
  | head::tail ->
    if target = head then List.rev (head::result)
    else if target > head then large_first tail (target - head)
(head::result)
    else large_first tail target result
;;
```

Counter example:

Input:

```
7
2 3 4 5 6
```

Output:

```
[]
```

Should output: `[2, 5]` or `[3, 4]`.

b. Proper solution

From wiki, the proper way to solve the knapsack (subset sum problem) is using dynamic programming to handle a two-dimension array. Since in OCaml lists are immutable, it is hard to perform dynamic programming. Therefore, this part is written in Python.

```
def knapsack(arr, target):    # dynamic programming
    arr_len = len(arr)
    subset = [[False for j in range(target + 1)] for i in range(arr_len + 1)]
    for i in range(arr_len + 1):
        subset[i][0] = True

    for i in range(1, arr_len + 1):
        for j in range(1, target + 1):
            if subset[i-1][j]:
                subset[i][j] = True
            else:
                if arr[i-1] > j:
                    subset[i][j] = False
                else:
                    subset[i][j] = subset[i-1][j-arr[i-1]]

    return subset

# To print the result, recursion can be used to judge each element.
def print_subset(subset, arr, target, n, result):
    if target == 0:
        print(sorted(result))
        return
    if n == 0 and target != 0 and subset[1][target]:
        result.append(arr[0])
        print(sorted(result))
        return
    if subset[n][target]:
        print_subset(subset, arr, target, n-1, result.copy())

    if target >= arr[n] and subset[n][target - arr[n]]:
        result.append(arr[n])
        print_subset(subset, arr, target - arr[n], n-1, result)


if __name__ == '__main__':
    target = int(input())
    arr = []
```

```
    for i in input().split():
        arr.append(int(i))
    subset = knapsack(arr, target)
    print_subset(subset, arr, target, len(arr) - 1, [])
```

3. Running time of three Knapsack algorithms

a. Smaller first

By choosing different sizes of arrays, the running time is shown below:

```
Size: 10    end
Execution time: 0.000013
Size: 20    end
Execution time: 0.000017
Size: 30    end
Execution time: 0.000030
Size: 40    end
Execution time: 0.000022
Size: 50    end
Execution time: 0.000011
Size: 100    end
Execution time: 0.000084
Size: 1000    end
Execution time: 0.000055
Size: 10000    end
Execution time: 0.000452
Size: 100000    end
Execution time: 0.003987
```

This algorithm is supposed to run in $\mathcal{O}(n)$ time, but due to extra computation like `if` statement and push the current element into the result list, according to the time complexity table, it is likely to have complexity $\mathcal{O}(n \log n)$

b. Larger first

Same as the above one, the output is shown below:

```
Size: 10    end
Execution time: 0.000006
Size: 20    end
Execution time: 0.000011
Size: 30    end
Execution time: 0.000009
Size: 40    end
Execution time: 0.000009
Size: 50    end
Execution time: 0.000059
Size: 100    end
Execution time: 0.000016
Size: 1000    end
Execution time: 0.000180
Size: 10000    end
Execution time: 0.000509
```

```
Size: 100000    end
Execution time: 0.003707
```

We can see that the running time of the larger-first algorithm differs a little with the smaller-first algorithm, but it in general costs less time a little.

c. Dynamic programming algorithm

I choose the target number to be a random number in $[100, 130]$, and the numbers in the test array are all range in $[0, 25]$.

The test program's code is:

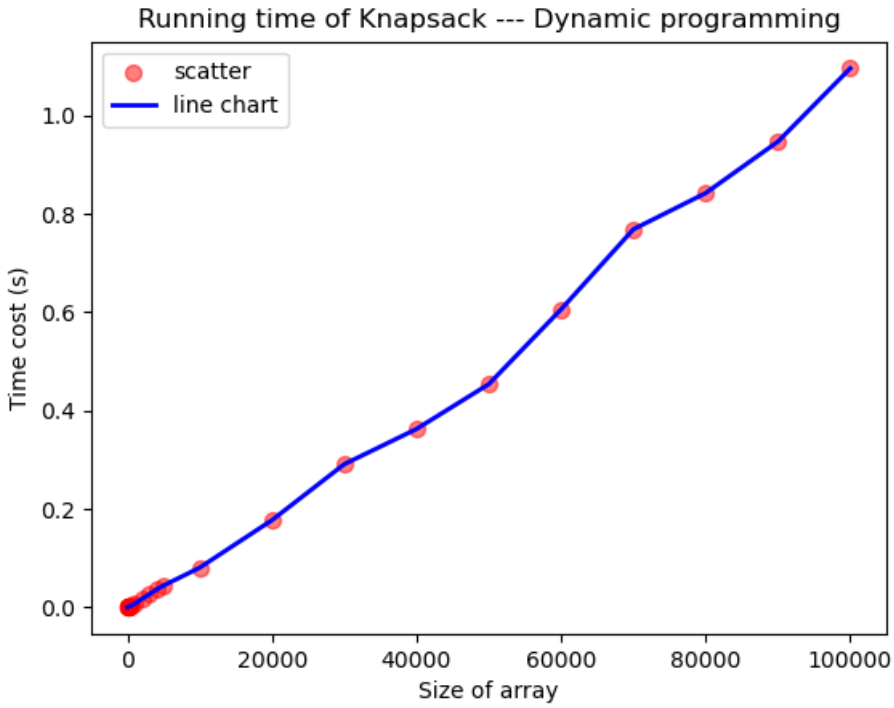
```
from knapsack import knapsack, print_subset
import random
import time
import matplotlib.pyplot as plt
import numpy


def create_list(s):
    result = []
    for i in range(s):
        result.append(random.randint(0, 25))
    return result


random.seed()
sizes = [10, 20, 30, 40, 50, 100, 200, 300, 400, 500, 1000, 2000, 3000,
4000, 5000,
         10000, 20000, 30000, 40000, 50000, 60000, 70000, 80000, 90000,
100000]
y = []
for size in sizes:
    arr = create_list(size)
    target = random.randint(100, 130)
    start = time.time()
    subset = knapsack(arr, target)
    end = time.time()
    # print_subset(subset, arr, target, size - 1, [])
    print("Size:", size, "Execution time:", end-start)
    y.append(end-start)

plt.figure()
scatter = plt.scatter(sizes, y, color='red', s=50, alpha=0.5)
plot ,= plt.plot(sizes, y, color="blue", linewidth=2.0, label="running
time")
plt.legend(handles=[scatter, plot], labels=["scatter", "line chart"])
plt.title("Running time of Knapsack --- Dynamic programming")
plt.xlabel("Size of array")
plt.ylabel("Time cost (s)")
plt.show()

plt.figure()
p1 = numpy.polyfit(x=sizes, y=y, deg=1)
yvals = numpy.polyval(p1, sizes)
plt.scatter(sizes, y, color="red", alpha=0.5, label="scatter")
```

```
plt.plot(sizes, yvals, color="blue", linewidth=2.0, label="linear fit")
plt.legend()
plt.title("Linear fit of output")
plt.xlabel("Size of array")
plt.ylabel("Time cost (s)")
plt.show()
```

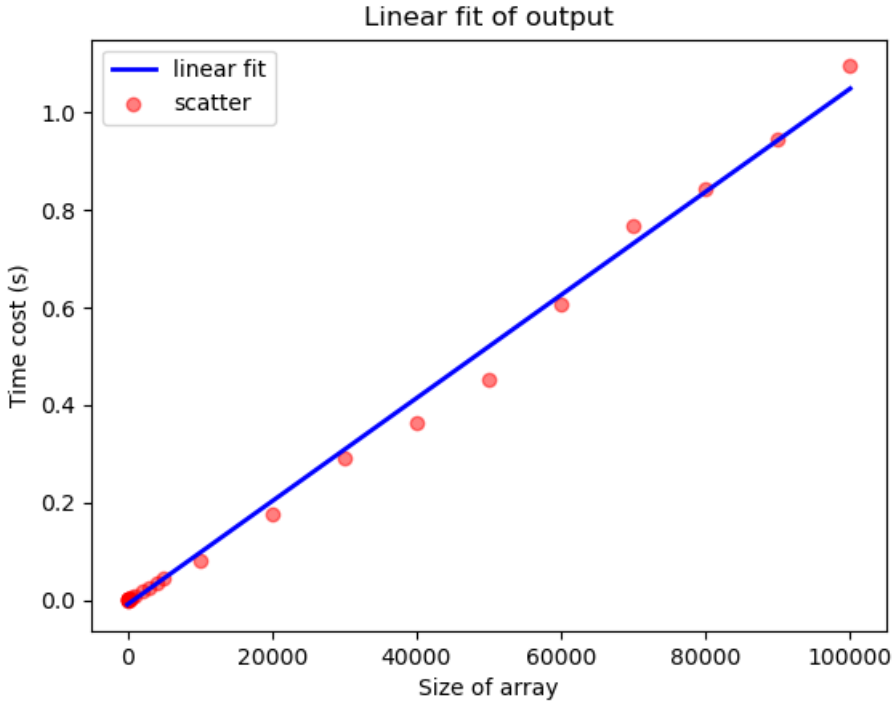
And the time output is:

```
Size: 10 Execution time: 0.0001468658447265625
Size: 20 Execution time: 0.00020837783813476562
Size: 30 Execution time: 0.0002999305725097656
Size: 40 Execution time: 0.00038504600524902344
Size: 50 Execution time: 0.00038623809814453125
Size: 100 Execution time: 0.0007736682891845703
Size: 200 Execution time: 0.0013849735260009766
Size: 300 Execution time: 0.0021965503692626953
Size: 400 Execution time: 0.002763509750366211
Size: 500 Execution time: 0.0036928653717041016
Size: 1000 Execution time: 0.0071218013763427734
Size: 2000 Execution time: 0.016954660415649414
Size: 3000 Execution time: 0.02609395980834961
Size: 4000 Execution time: 0.03602409362792969
Size: 5000 Execution time: 0.04427981376647949
Size: 10000 Execution time: 0.08069109916687012
Size: 20000 Execution time: 0.1772470474243164
Size: 30000 Execution time: 0.29102230072021484
Size: 40000 Execution time: 0.3626530170440674
Size: 50000 Execution time: 0.4538257122039795
Size: 60000 Execution time: 0.6058144569396973
Size: 70000 Execution time: 0.7685534954071045
Size: 80000 Execution time: 0.8421218395233154
Size: 90000 Execution time: 0.9468364715576172
Size: 100000 Execution time: 1.0956668853759766
```

Using `matplotlib.pyplot`, we can show the scatter and line chart of the data:



Then it comes the linear fit of the obtained points:



Theoretically, the dynamic programming algorithm should have the time complexity of $\mathcal{O}(TN)$, where T is the **target number** and **N** is the size of the input array. From the plot generated, we can obviously find out that the time increase is linear, which conforms with the theory. However, due to the target number's size, the time cost will differ.

Since the smaller-first and larger-first algorithm just traverse the test array once, they will be much faster than the dynamic programming algorithm. However, as they cannot give the right answer, only serves as a greedy algorithm, the time cost comparison will have no significance.

4. Topological sort

Topological sort of a **directed graph** is a linear ordering of its vertices such that for every directed edge uv from vertex u to vertex v, u comes before v in the ordering.

OCaml code is shown below:

```
let create_hash hash n =
  for i = 0 to n - 1 do
    Hashtbl.add hash i []
  done
;;

let init hash e =     (*Initialize hashtable according to input*)
  for i = 0 to e - 1 do
    let (node_from, node_to) = Scanf.scanf " %d %d" (fun a b -> (a, b)) in
    let x = Hashtbl.find hash node_from in
    Hashtbl.remove hash node_from;
    if List.mem node_to x then Hashtbl.add hash node_from x
    else Hashtbl.add hash node_from (node_to :: x)
  done
;;

let dfs graph visited start =     (*Use dfs to find the path*)
  let rec explore path visited node =
    if List.mem node visited then visited
    else
      let new_path = node :: path in
      let edges = List.assoc node graph in
      let visited = List.fold_left (explore new_path) visited edges in
      node :: visited
  in explore [] visited start
;;

let toposort graph =
  List.fold_left (fun visited (node,_) -> dfs graph visited node) [] graph
;;

let rec print l =
  match l with
  | [] -> print_string ""
  | head :: [] -> print_int head
  | head :: tail -> print_int head; print_string " "; print tail
;;
let n = read_int();;
let h = Hashtbl.create n;;
let e = read_int();;

create_hash h n;;
init h e;;
let l = Hashtbl.fold (fun x y z -> (x, y) :: z) h [];;
(*Transform hashtable into list*)
let result = toposort l;;
print result;;
```