CS2040 AY19/20 ST2 Midterm Solutions

CS2040 Teaching Staff

July 2020

Q1. The Museum of Valuable Liquids.

You have N liquids to take, each with $Weight[i]$ kg available to be taken, and total value \$ $Value[i]$ for the entire weight. You only can carry a maximum of C kg of liquids. For every liquid, you can choose to take none of it, part of it, or the entire amount.

What is the maximum value can you carry out?

(Note: This is known as the *fractional knapsack problem.*)

(a) Algorithm

Algorithm 1 Compute maximum value of items you can bring out, for max capacity C

```
function EXTRACT(C, N, Weight[1..N], Value[1..N])
    Items ← [1..N]
    SORTDECREASING(Items, item → Value[item]/Weight[item])    ▷ Sort by value per unit weight
    val ← 0
    for all item ∈ Items do                    ▷ Take items in decreasing value/weight ratio
        if C ≤ 0 then break                    ▷ Stop if already full
        maxAllowed ← min(Weight[item], C)      ▷ Take as much as possible
        C ← C − maxAllowed
        val ← val + maxAllowed × (Value[item]/Weight[item])
    end for
    return val
end function
```

(b) Runtime Analysis

Before the loop, we initialize some variables. The first $Items \leftarrow [1..N]$ line takes $O(N)$ time to create the array. Then, calling SORTDECREASING takes $O(N \log N)$, as our comparison function runs in $O(1)$. The remainder of the statements ($val \leftarrow 0$ and **return** val) are $O(1)$. Thus, everything outside of the loop runs in $O(N \log N)$.

Inside the loop body, we have a constant number of arithmetic operations, and so this runs in $O(1)$. The loop runs for at most N iterations, so the whole loop runs in $O(N)$.

Thus, the total time our algorithm takes is $O(N \log N) + O(N) = O(N \log N)$.

(c) Optimal Answer

Assume for simplicity, that all liquids' value-to-weight ratios are different. (If two liquids have the same ratio, we can combine them and treat them as the same liquid.)

Let's say we have some different answer OTHER, that picks different amounts of different liquids from our EXTRACT algorithm, and gets **the optimal value**, better than that of EXTRACT. We order the choices of liquids made, by decreasing value-to-weight ratio.

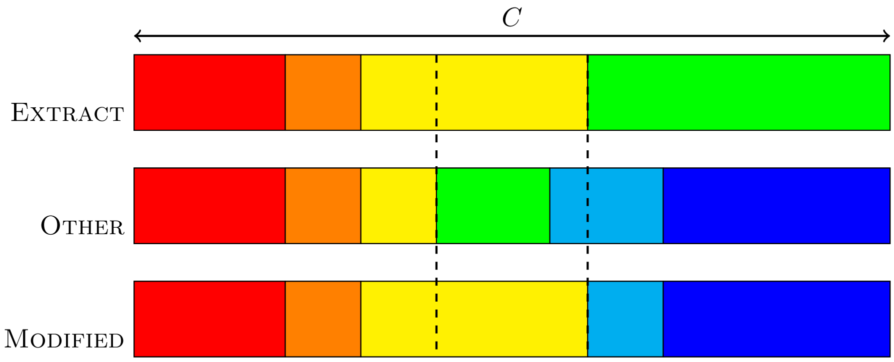

Figure 1: Redder hues indicate higher value/weight ratio.

As Extract and Other are different, there is some liquid L with the best value/weight ratio, that Extract chose fully, while Other chose some lower value/weight liquids for.

We create a Modified solution almost identically to Other, but for that segment of capacity, we copy Extract's choices and choose L instead. In that segment, the overall value/weight ratio strictly increases compared to Other. Now, Modified is a solution better than Other, but we originally assume that Other was the best! Hence we derive a contradiction, and we cannot do better than Extract.

(Extract is what we call a **greedy algorithm**, where we repeatedly take the 'best right now' choice at all times. This form of optimality proof is called an **exchange** argument. Given a supposedly optimal solution, we 'exchange' part of that solution for the corresponding part of our solution, and improve it further, which contradicts the supposed optimality.)

Q2. Duplicate Removal

Given an (potentially unsorted) array of n integers, return a duplicate-free version of the array.

(Note: The question stated 'duplicate-free list', and did not state clearly that you cannot lose any elements.)

(a) $O(n \log n)$

Sort the array so that all copies of the same element are adjacent to each other, then do a linear pass over the array to remove the duplicates.

Algorithm 2 Duplicate Removal in $O(n \log n)$

function RemoveDuplicates(n, $A[1..n]$)
    SortIncreasing(A)
    $output \leftarrow []$
    Append $A[1]$ to *output*
    for $i \leftarrow 2$ to n **do**
        if $A[i] \neq A[i-1]$ **then**
            Append $A[i]$ to *output*
        end if
    end for
    return *output*
end function

We sort all the items in $O(n \log n)$, then scan the sorted array in $O(n)$. By sorting, all duplicate items are grouped together in a contiguous chunk in the array. Then, whenever we see two adjacent items with distinct values, the second one is the head of a new chunk, and so we add it as a new distinct item.

One implicit assumption, is that comparison is $O(1)$ time.

(b) $O(n)$ average/w.h.p

Make a linear pass over the array, and keep track of the elements seen in a hash table to detect duplicates.

Algorithm 3 Duplicate Removal in expected $O(n)$

```
function REMOVEDUPLICATES(n, A[1..n])
    H ← empty hash table
    output ← []
    for i ← 1 to N do
        if A[i] is not in H then
            Insert A[i] into H
            Append A[i] to output
        end if
    end for
    return output
end function
```

This solution assumes that hashtable operations all run in amortized $O(1)$, instead of the possible worst-case time of $O(n)$.

(Extra out-of-syllabus notes: If we have a fixed hash function like in Java, then there is always a worst case input with all items colliding, causing hashtable insert to be $O(n)$. Hence, hashtable analysis is actually done with choosing a random hash function, from a family of possible hash functions. Hence, 'with high probability', we get a even distribution of keys. This concept will be explored more in CS3230/5330.)

Partial Credits

- **Radix Sort**

  Radix sort runs in $O(nd)$, where d is the maximum 'digit length' of the keys to sort. However, we did not give any bound on the range of integers to compare, so you **must** declare the assumption that d is a small constant that can be absorbed into the O.

  Even so, as long the maximum value is at least n, then we have at least $\log_b n$ digits (in base b). Then, asymptotically, radix sort runs in $O(n \log_b n) = O(n \log n)$, and does not perform better than the comparison sorts.

Q3. Merging Binary Heaps

(a) $O(n + m)$

Combine the elements in both of the heaps into a single array, and then run heapify.

Algorithm 4 Merge Binary Heaps in $O(n + m)$

```
function MERGEHEAP(n, h1, m, h2)
    h ← []
    Add all elements in h1 to h
    Add all elements in h2 to h
    HEAPIFY(h)
    return h
end function
```

Assuming both heaps can be converted into flat arrays with no structure (e.g. for binary heaps, we return the inner array), we can simply combine these two arrays, and perform MAKEHEAP/HEAPIFY on the combined array. This takes $O(n + m)$ for both concatenating the arrays, and MAKEHEAP.

(b) $O(n \log m)$

Another method to combine heaps, is to treat only one side as a bag of elements of size n, and repeatedly INSERT them into the other heap of size m.

Algorithm 5 Merge Binary Heaps in $O(n \log m)$

> **function** MERGEHEAP(n, h_1, m, h_2)
>   **for all** $item \in h_1$ **do**
>   Insert $item$ into h_2
>   **end for**
>   **return** h_2
> **end function**

(c) $O(\min\{n + m, n \log m, m \log n\})$

We first compute all of $n + m$, $n \log m$, $m \log n$, and take the minimum of them.

- If the minimum is $n + m$, we make use of the strategy in (a), where we call MAKEHEAP on the combined array.
- Otherwise, if the minimum is $n \log m$, we use the strategy in (b), and repeatedly INSERT each of the size-n heap's elements into the size-m heap.
- A similar idea applies to the case where $m \log n$ is minimum, and we use (b)'s strategy again, but exchanging the roles of the size-m and size-n heaps.

Analysis: (Here, $\log n$ will refer to the base-2 logarithm.)
The initial arithmetic involves a constant number of arithmetic operations, and so is $O(1)$ time. In other words, the maximum time is at most some constant k_0.
Now, we analyse the time taken for the chosen strategy:

- Minimum is $n + m$:

  This runs in $f(n, m) \in O(n + m)$ time. In other words, we have some constant k_1, such that $f(n, m) \leq k_1(n + m)$ for large enough n, m.
- Minimum is $n \log m$:

  This runs in $g(n, m) \in O(n \log m)$ time. In other words, we have some constant k_2, such that $g(n, m) \leq k_2(n \log m)$ for large enough n, m.
- Minimum is $m \log n$:

  Similarly, this case runs in $g(m, n) \leq k_2(m \log n)$ for large enough m, n.

Let $k = \max(k_0, k_1, k_2)$. Then, assume m, n are greater than 2 (so that $\log m, \log n > 1$). Then, we can say that the overall algorithm runs in:

$$\begin{aligned}
h(m,n) &\leq k_0 + \begin{cases} k_1(n+m) & (n + m \leq n \log m, m \log n) \\ k_2(n \log m) & (n \log m \leq n + m, m \log n) \\ k_2(m \log n) & (m \log n \leq n + m, n \log m) \end{cases} \\
&\leq k + k \begin{cases} n+m & (n + m \leq n \log m, m \log n) \\ n \log m & (n \log m \leq n + m, m \log n) \\ m \log n & (m \log n \leq n + m, n \log m) \end{cases} && (k \geq k_0, k_1, k_2) \\
&= k + k \min\{n + m, n \log m, m \log n\} && \text{(Definition of min)} \\
&\leq 2k \min\{n + m, n \log m, m \log n\} && (n, m > 2 \implies \min\{\ldots\} \geq 1)
\end{aligned}$$

Hence, our overall time complexity is $O(\min\{n + m, n \log m, m \log n\})$.
(This method works for any finite number of possible choices/strategies.)

Q4. Buggy Code: Exploring Planet Nine

(a) LandingVehicle

```java
public class LandingVehicle {

        public String name;
        public int fuel;

        LandingVehicle(){
        }

        public void setName(String n){
                name = n;
        }

        public void dispatch(){
        }

        public boolean testVehicle(int i){
                for (int wheel = 0; wheel < 4; wheel++){
                        testWheel(wheel);
                }
                return true;
        }

        public void testWheel(int w) {
                boolean failed = true;
                for (int i=10; i>0; i--){
                    // Do test.
                }
                if (failed) return (w / i);
        }

}
```

In `public void testWheel(int w)`, we have a for loop that **declares** and initializes `i=10`, and decrements `i` as long as it is strictly positive. However, as `i` is declared only within the scope of the for loop, `w / i` is not valid.

Furthermore, even if `i` was declared outside the loop, there is still one issue. If the loop does not break midway, the final value of `i` would be 0 after the decrement, and this would result in division by zero.

Also, the return type for `public void testWheel(int w)` is `void`. It should not `return (w / i);`.

(b) `Rover`

```
1  public class Rover {
2
3          public static int roverCount = 0;
4          public String pilot;
5
6          public Rover(String p){
7                  pilot = p;
8          }
9
10         public Rover() {
11                 roverCount++;
12         }
13
14         public boolean testVehicle(int i){
15                 for (int wheel = 0; wheel < 4; wheel++){
16                         // do test;
17                 }
18                 return true;
19         }
20
21         public static int analyzeRovers(){
22                 for (int i=0; i<roverCount; i++){
23                         if (testVehicle(i)){
24                                 return -1;
25                         }
26                 }
27                 return 1;
28         }
29
30 }
```

In line 23, within `static int analyzeRovers()`, we call `boolean testVehicle(int)`. However, as `analyzeRovers()` is a static method, it needs a `Rover` instance to call the instance method `testVehicle(int)`. This is not the case in line 23.

(c) Probe

```java
public class Probe {

    public String name;

    public Probe(String n){
        name = n;
    }

    public String checkFuel(){
        Rover rover = new Rover();
        if (rover.testVehicle()) return "Ok" else return "Nope!";
    }
}
```

Firstly, `Rover.testVehicle(int)` requires an `int` parameter. Then, we need a semicolon before `else`:

```java
if (rover.testVehicle(0)) return "Ok"; else return "Nope!";
```

(d) NinthPlanet

```java
1 public class NinthPlanet {
2 
3     public LandingVehicle[] rovers;
4 
5     public NinthPlanet(String[] names){
6         int numRovers = names.length;
7         rovers = new Rover[numRovers];
8         for (int i=0; i<numRovers; i++){
9             rovers[i].setName(names[i]);
10         }
11     }
12 
13     public int dispatchRovers() {
14         for (int i=0; i<rovers.length; i++){
15             rovers[i].dispatch();
16         }
17         return 17;
18     }
19 }
```

On line 7, we initialize `rovers = new Rover[numRovers]`, in the constructor of `NinthPlanet`. This creates a new array, of `numRover` references to `Rover` objects, each initialized to the `null` reference.

However, on line 9, we immediately call `rovers[i].setName(names[i]);` without actually having a valid `Rover` object at that slot. This will result in a `NullPointerException`.

Also, as the class `Rover` is not a subclass of `LandingVehicle`, `rovers = new Rover[numRovers];` will result in a compilation error.

Q5. Linear Data Structures

(a) `splitIntoTwo(a,b,c)`

```
void splitIntoTwo(LinkList a, LinkList b, LinkList c) {
    int sizeOfB = (a.num() + 1) / 2;
    for (int i = 0; i < sizeOfB; i++) {
        b.append(a.peekHead());
        a.deleteHead();
    }
    while (a.num() > 0) {
        c.append(a.peekHead());
        a.deleteHead();
    }
}
```

(b) Single Pointer to `head`

Our goal is to implement a *circular doubly linked list* in this problem, which is simply a doubly linked list where the first and last nodes in the linked list are linked by pointers. This will then allow us to implement `append(i)` and `deleteTail()` without the presence of a `head` pointer.

- `prepend(i)`
  Assume that the node to be inserted is `node`
  - Linked list is empty: `node.next` is set to `node`, `node.prev` is set to `node`, `head` is set to `node`
  - Linked list is not empty: `node.next` is set to `head.next`, `node.prev` is set to `head.prev`, `head.prev.next` is set to `node`, `head.prev` is set to `node`, `head` is set to `node`
- `append(i)`
  Assume that the node to be inserted is `node`
  - Linked list is empty: `node.next` is set to `node`, `node.prev` is set to `node`, `head` is set to `node`
  - Linked list is not empty: `node.next` is set to `head`, `node.prev` is set to `head.prev`, `head.prev.next` is set to `node`, `head.prev` is set to `node`
- `deleteHead()`
  - Linked list is empty: operation fails
  - Linked list has size 1: `head` is set to `null`
  - Linked list has size > 1: `head.prev.next` is set to `head.next`, `head.next.prev` is set to `head.prev`, `head` is set to `head.next`
- `deleteTail()`
  - Linked list is empty: operation fails
  - Linked list has size 1: `head` is set to `null`
  - Linked list has size > 1: `head.prev.prev.next` is set to `head`, `head.prev` is set to `head.prev.prev`