

Solutions by Artem Gritsenko, Ahmedul Kabir, Yun Lu, and Prof. Ruiz

**Problem I.** By Artem Gritsenko, Yun Lu, and Prof. Ruiz**a. Algorithm.**KNAPSACK ( $n, W, w_1, \dots, w_n, v_1, \dots, v_n$ )#  $W$  is the knapsack capacity#  $n$  is the number of items that can be packed in the knapsack# item  $i$  has weight  $w_i$  and value  $v_i$ # Assume that  $w_1 \leq w_2 \leq \dots \leq w_n$  and  $v_1 \geq v_2 \geq \dots \geq v_n$ # We'll use two variables: a set  $S$  that contains the items selected to be packed in the# knapsack, and  $w$  which is the sum of the weights of the items in  $S$ . $w = 0$  $S = \emptyset$  $i = 1$ while ( $i \leq n$ ) and ( $w + w_i \leq W$ ) $w = w + w_i$  $S = S \cup \{i\}$  $i = i + 1$ return  $S$ **b. Complexity analysis.**

The algorithm runs in linear time in the number of items ( $= n$ ). In the worst case, it will traverse the full list of  $n$  items in the while loop. Processing each individual item (loop body) takes constant time. Hence,  $T(n) = O(n)$ .

**c. Algorithm Correctness.**

First we show that it terminates. The while loop considers each item at most once, and hence the algorithm terminates after at most  $n$  iterations.

Now we prove that the algorithm terminates with a correct answer. That is, that the set  $S$  output by the algorithm is an optimal solution. In other words, we need to prove that:

1.  $S$  is a solution: The sum of the weights of the items in  $S$  is less than or equal to  $W$ . This is true because an item  $i$  is added to  $S$  only if the current weight of  $S$  together with  $w_i$  doesn't exceed  $W$ .

2.  $S$  is an optimal solution: For any other solution  $S'$ , the sum of the values of the items in  $S$  is greater than or equal to the sum of the values of the items in  $S'$ .

Let's prove that the set  $S$  output by the algorithm is optimal using a proof by contradiction. Assume that  $S$  is not optimal. Then there is another set  $O$  which is an optimal solution. Let's sort the items in  $S$  and the items in  $O$  in increasing order by weight (= decreasing order by value).

$$S = \{i_1, i_2, \dots, i_k\}$$

$$O = \{j_1, j_2, \dots, j_m\}$$

Let  $r \geq 0$  be the first index in which the two sets differ. That is,

$$S = \{i_1, i_2, \dots, i_{r-1}, i_r, i_{r+1}, \dots, i_k\}$$

$$O = \{j_1, j_2, \dots, j_{r-1}, j_r, j_{r+1}, \dots, j_m\}$$

with  $i_1 = j_1, i_2 = j_2, \dots, i_{r-1} = j_{r-1}$ , and  $i_r \neq j_r$ .

Since our algorithm chose  $i_r$  in such a way that  $i_r$  is the item with smaller weight and higher value among all items not in  $i_1, \dots, i_{r-1}$ , then:

- the weight of  $i_r$  is less than or equal to the weight of  $j_r$ , and
- the value of  $i_r$  is greater than or equal to the value of  $j_r$ .

Then we use the *exchange argument*: we can replace  $j_r$  with  $i_r$  in  $O$ , without exceeding the weight limit, and keeping the value of  $O$  the same or increasing it. If the value of  $O$  increases, then this yields a contradiction as we assumed that  $O$  was optimal. If the value of  $O$  doesn't increase, then we repeat the same exchange argument with the next item, the  $(r + 1)^{st}$  item. We continue to do so until we transform  $O$  into a solution with higher value, which would contradict the assumption that it was optimal; or we make it equal to  $S$  without increasing its weight, which would prove that  $S$  is an optimal solution too.

## **Problem II. By Ahmedul Kabir and Prof. Ruiz**

### **a. Pseudocode for Offline Caching problem**

```
FurthestInFuture(<r1, r2, ... rn>, k):  
    for i = 1 to n  
        if ri is in cache:  
            then print "Cache Hit"  
        else  
            print "Cache Miss"  
            if cache is not full  
                Add ri to cache  
            else (if cache is full)  
                // Find element in cache with furthest distance  
                distfurthest = 0  
                for j = 1 to k  
                    p = i + 1 // p is an index on the list of requests  
                    Keep increasing p until rp == cache[j]  
                    if p reaches end of sequence without finding rp  
                        then  
                            distj = infinity  
                            furthest = j  
                            break from the inner for loop  
                else  
                    distj = p - i  
                if distj > distfurthest  
                    then  
                        furthest = j  
                        distfurthest = distj  
  
                Evict Cache[furthest]  
                Add ri to cache
```

Kabir's Python implementation of this pseudo-code is available at:

<http://web.cs.wpi.edu/~cs2223/b13/HW/HW5/Solutions>

## Running time of the algorithm

The outer loop runs  $n$  times, and the inner loop  $k$  times. For each iteration of the inner loop, we may have to traverse  $(n - i)$  items at worst. So the running time is

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

### b. The off-line caching problem exhibits optimal substructure.

A problem exhibits optimal substructure if an optimal solution to the problem contains within it optimal solutions to subproblems.

Let  $R = \{r_1, r_2, \dots, r_n\}$  be a list of requests, and  $k$  the cache of size. Let also  $E = \{e_1, e_2, \dots, e_n\}$  be a list of evictions, where each  $e_i$  is either “cache hit”, if  $r_i$  is in the cache at time  $i$ , or the name of the cache element to be evicted at time  $i$  because of a cache miss occurred at that time. Let's assume that  $E$  is optimal for  $R$  and  $k$ . That is,  $E$  contains the minimum possible number of evictions (= cache misses) for the list of requests  $R$  and a cache of size  $k$ .

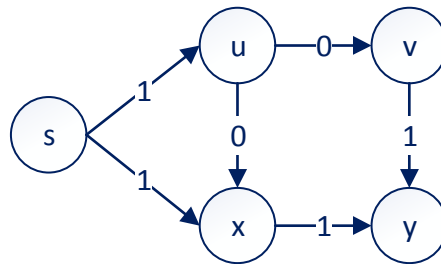
In order to show that the off-line caching problem exhibits optimal substructure, we need to show that any subsequence of  $E$ ,  $E_i = \{e_i, e_{i+1}, \dots, e_n\}$  is optimal for the sublist of requests  $R_i = \{r_i, r_{i+1}, \dots, r_n\}$  of  $R$  starting with the cache content at time  $i$ . It suffices to show that this is true for  $i$  equal to the time when the first cache miss occurred, and then apply the same argument for other cache misses afterwards. So let  $i$ ,  $1 \leq i \leq n$ , be the first time when a cache miss occurred. Assume by way of contradiction that the solution  $E_i = \{e_i, e_{i+1}, \dots, e_n\}$  is NOT optimal for the sublist of requests  $R_i = \{r_i, r_{i+1}, \dots, r_n\}$  starting with the cache content at time  $i$  right when the cache miss was detected. Hence, there must be another solution  $O = \{o_i, o_{i+1}, \dots, o_n\}$  which incurs in fewer cache misses on  $R_i = \{r_i, r_{i+1}, \dots, r_n\}$  than  $E_i$ , starting with the same cache content that  $E_i$  does at time  $i$ . In that case, the list of evictions  $New = \{e_1, e_2, \dots, e_{i-1}, o_i, o_{i+1}, \dots, o_n\}$  is a solution for  $R$  with fewer cache misses than  $E$ . This is a contradiction, as  $E$  is optimal for  $R$ . Hence, as claimed,  $E_i$  is optimal for  $R_i$ .

- c. The furthest-in-future strategy produces the minimum possible number of cache misses. See a proof of this fact in the slides of Chapter 4 of Jon Kleinberg's and Éva Tardos' Algorithm Design textbook. Available at Prof. Kevin Wayne's Algorithms Course Webpage at Princeton University. See Greedy Algorithms Part I slides 34-43 at <http://www.cs.princeton.edu/~wayne/kleinberg-tardos/pdf/04GreedyAlgorithmsI.pdf>.

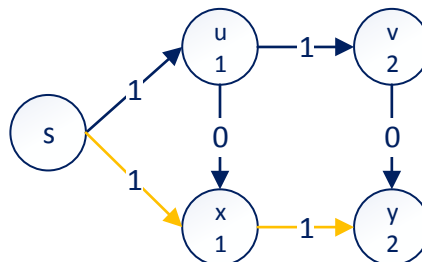
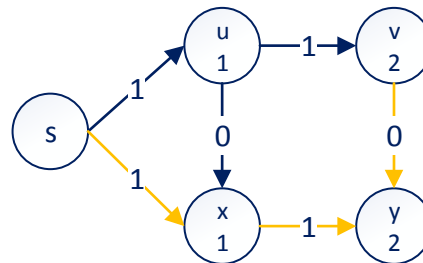
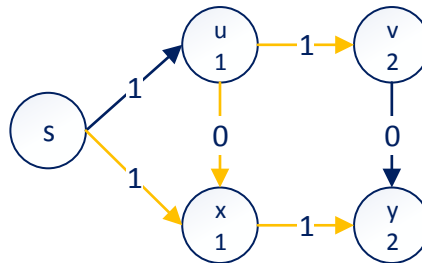
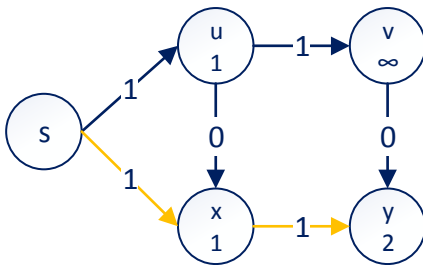
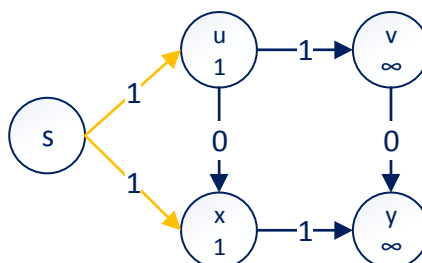
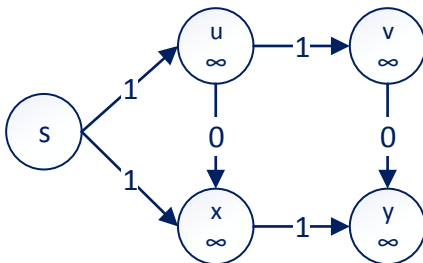
### **Problem III.** By Yun Lu.

The key idea of this problem is to find a graph that has at least two optimal paths, which of course will have the same weight. Furthermore, we'll construct these two paths in such a way that Dijkstra's algorithm will relax the edges of one of the paths in the order in which they occur in the path, but will not do so for the other path.

Consider the following example  $G = (V, E)$  with  $V = \{s, u, v, x, y\}$  and  $E = \{(s, x), (s, u), (u, v), (u, x), (v, y), (x, y)\}$ , where the weights of these edges are given below:



Applying Dijkstra's algorithm to the above graph:



In the execution of Dijkstra's algorithm above, the edges are relaxed (in yellow) in the following order:  $(s,x)$ ,  $(s,u)$ ,  $(x,y)$ ,  $(u,x)$ ,  $(u,v)$ ,  $(v,y)$ .

Note that there are three paths from  $s$  to  $y$ : 
$$\begin{cases} s \rightarrow u \rightarrow v \rightarrow y \\ s \rightarrow u \rightarrow x \rightarrow y \\ s \rightarrow x \rightarrow y \end{cases}$$

all of them with weight 2, hence all three paths are optimal. The edges in the path  $s \rightarrow u \rightarrow x \rightarrow y$  are relaxed out of order since  $(x,y)$  is relaxed before  $(u,x)$ .