Lab03-Greedy Strategy

CS214-Algorithm and Complexity, Xiaofeng Gao, Spring 2021.

∗ If there is any problem, please contact TA Haolin Zhou.
∗ Name: Log Creative Student ID: Email: logcreative-lzl@sjtu.edu.cn

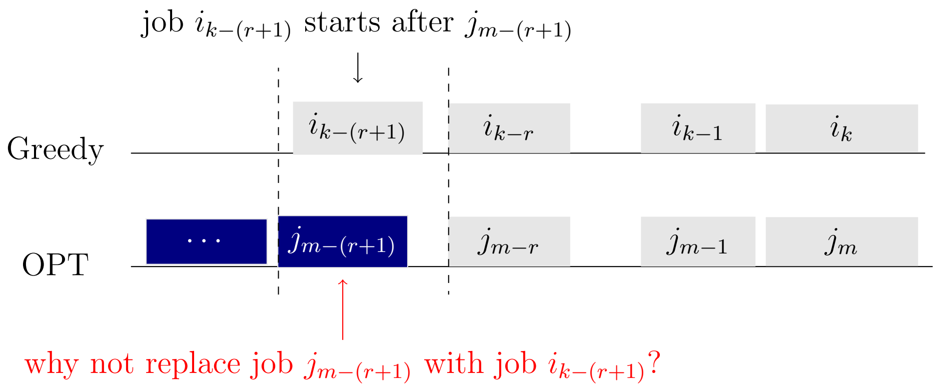
1. *Interval Scheduling.* Interval Scheduling is a classic problem solved by **greedy algorithm**: given n jobs and the j-th job starts at s_j and finishes at f_j. Two jobs are compatible if they do not overlap. The goal is to find maximum subset of mutually compatible jobs. Tim wants to solve it by sort the jobs in descending order of s_j. Is this attempt correct? Prove the correctness of such idea, or else provide a counter-example.

   Solution. Tim is correct. Prove this idea by contradiction.

   Assume greedy is not optimal for scheduling the jobs in descending order of s_j.

   Let $i_1, i_2, \cdots, i_k$ denote set of jobs selected by greedy.

   Let $j_1, j_2, \cdots, j_m$ denote set of jobs in an optimal solution with $i_k = j_m, i_{k-1} = j_{m-1}, \cdots, i_{k-r} = j_{m-r}$ for the largest possible value of r.


job $i_{k-(r+1)}$ starts after $j_{m-(r+1)}$
Greedy
$i_{k-(r+1)}$
i_{k-r}
i_{k-1}
i_k
OPT
$\cdots$
$j_{m-(r+1)}$
j_{m-r}
j_{m-1}
j_m
why not replace job $j_{m-(r+1)}$ with job $i_{k-(r+1)}$?

   For job $i_{k-(r+1)}$, it starts after $j_{m-(r+1)}$. But why not replace job $j_{m-(r+1)}$ with job $i_{k-(r+1)}$, so that the solution is still feasible and optimal. Then, it contradicts the maximality of r.

   So, the greedy solution is optimal in this case. □

2. *Done deal.* In a basketball league, teams need to complete player trades through matching contracts. Every player is offered a contract. For the sake of simplicity, we assume that the unit is M, and the size of all contracts are integers. The process of contract matching refers to the equation: $\sum_{i\in A} a_i = \sum_{j\in B} b_j$, where a_i refers to the contract value of player i in team A involved in the trade and b_j refers to the value of player j in team B.

   Assume that you are a manager of a basketball team and you want to get **one** star player from another team through trade. The contract of the star player is $n (n \in \mathbb{N}^+)$. The goal is to complete the trade with as few players as possible.

   (a) Describe a **greedy** algorithm to get the deal done with the least players in your team. Assume that there are only 4 types of contracts in your team: $25M$, $10M$, $5M$, $1M$, and there is no limit to the number of players. Prove that your algorithm yields an optimal solution.

   (b) Suppose that the available contract sizes are powers of c, i.e., the values are $c^0, c^1, \ldots, c^k$ for some integers $c > 1$ and $k \geq 1$. Show that the greedy algorithm always yields an optimal solution.

(c) Give a set of contract sizes for which the greedy algorithm does not yield an optimal solution. Your set should include a $1M$ so that there is a solution for every value of n.

Solution. Explain the greedy algorithm for all three sub-problems. The goal is to find the mimimum size of the collection of contract values such that

$$\sum_{i \in A} a_i = n$$

Firstly, **to avoid the waste on the expenses** (beacuse there is no limit to the number of players), the summation of values of the trade players should be exactly n. Otherwise, just use the biggest type of contract will lead to the optimal solution of one player.

Then, try to use the maximum value of the type of contract to reduce the remaining value, for each *trading round.*

Algorithm 1: The greedy algorithm for matching contracts

```
Input: The contract of the star player $n$, contract type array $\{x_i\}_{x=1}^{t}$
Output: The required amount $N$ of players for the deal
1 Sort $\{x_i\}_{i=1}^{n}$ in a descending order;
2 $N \leftarrow 0$;
3 for $i \leftarrow 1$ to $t$ do
4     $N \leftarrow N + \lfloor \frac{n}{x_i} \rfloor$;
5     $n \leftarrow n + (n \mod x_i)$;
6     if $n = 0$ then break;
7 if $n > 0$ then
8     return No greedy solution;
9 return $N$;
```

(a) **Case 1: Exchange exactly.** If an optimal solution tries to exchange the large size into the small size ones for one trading round, because

$$\begin{aligned} 25M &= 10M \times 2 + 5M \\ 10M &= 5M \times 2 \\ 5M &= 1M \times 5 \end{aligned}$$

yields an optimal solution for the exchange. As is noticed, the required amount of players is bigger than 1. In other word, 2 smaller contracts are least needed to meet the large expense. And the remaining part is structed to use the biggest types of contracts as the greedy algorithm goes, there is no optimization space to reduce the amount of players. So it is not an optimal solution, in order not to waste surplus money.

Case 2: Exchange not exactly. Maybe there may be a not exactly exchange the large size scenario. This will only happen when **2 sub-size is not enough**, specifically, $25M$ and $5M$.

Case 2a: The least size bigger than $1M$. For the least size bigger than $1M$, say $5M$, if the part cannot be exchanged by $5M$ (for example, $4M$), then it cannot be done by its multiplier as well.

$$x \nmid r \Rightarrow kx \nmid r$$

where $k \in \mathbb{N}_+, r < x$. Then,

$$x \nmid r + mx$$
$$kx \nmid r + mx$$

where $m \in \mathbb{N}_+$. So r has to be regarded seperately: r has to be consummed by $1M$.

Case 2b: The larger one. For $25M$, if it is first exchanged by $10M \times 2$, maybe the remaining part is $5M$ additional to $25M$, then one more $10M$ is required. But this takes at least 3 players instead of 2:

$$25M + 5M = 10M \times 3$$

If the remaining part is $6M$ or $4M$, according to Case 2a, you cannot deal with $r = 1M$ or $r = 4M$ with an optimized allocation, and the bigger part is already allocated by the greedy way:

$$25M + 5M + 1M = 10M \times 3 + 1M$$
$$25M + 1M \times 4 = 10M \times 2 + 5M + 1M \times 4$$

So the greedy algorithm is the best solution.

(b) Notice the neighbored types:

$$c^k = c^{k-1} \times c$$

where $c > 1, c \in \mathbb{N}$ and $k \geq 1$.

Case 1: $c = 2$ Similar to the exactly exchanging scenario, if the optimized method exchange the bigger one with the sub-small one, the amount of required players is definate to be more than the greedy one.

Case 2: $c > 2$

$$c^k > c^{k-1} \times 2$$

In one optimization block, where there is a remainder $r < c$ to be optimized (if $r \geq c$, it should be divided into the smaller-than-c part and bigger part first, because the bigger part is always better to be consummed by the larger type),

$$c^k + r = c^{k-1} \times 2 + c^{k-1} \times (c - 2) + r$$

because $c \nmid r$, r has to be comsummed by $1M$. In fact, this also holds for $c = 2$. The greedy solution is the optimal solution.

(c) The collection of unstatisfied sets is $\mathbf{U} = \{S | \exists x_i, x_{i+1} \in S : x_{i+1} < 2x_i\}$.

One example when one type is not two times bigger than the previous type: $1M, 7M, 10M$, To get $14M$,

$$10M + 1M \times 4 = 7M \times 2$$

The right one is better.

$\square$

3. *Set Cover.* **Set Cover** is a typical kind of problems that can be solved by greedy strategy. One version is that: Given n points on a straight line, denoted as $\{x_i\}_{i=1}^n$, and we intend to use minimum number of closed intervals with fixed length k to cover these n points.

(a) Please design an algorithm based on **greedy** strategy to solve the above problem, in the form of *pseudo code*. Then please analyze its *worst-case* complexity.

(b) Please prove the correctness of your algorithm.

(c) Please complete the provided source code by C/C++ (The source code *Code-SetCover.cpp* is attached on the course webpage), and please write down the output result by testing the following inputs:

i. the number of points $n = 7$;

ii. the coordinates of points $x = \{1, 2, 3, 4, 5, 6, -2\}$;

iii. the length of intervals $k = 3$.

Remark: Screenshots of running results are also acceptable

Solution. (a) Each time cover the largest amount of elements at the moment. In order to cover the maximum amount of the elements, the start of the interval is one of the existing element, which is the same as to make the end of the interval to be one of the element. The greedy algorithm is designed as follows:

Algorithm 2: Greedy Set Cover Algorithm

```
Input: An array $\{x_i\}_{i=1}^n$ of size $n$, the length of intervals $k = 3$
Output: The minimum number of closed intervals of length $k$ to cover the
        array points
Sort the array $\{x_i\}_{i=1}^n$;
$N \leftarrow 0$;
while $n > 0$ do
    $maxCov \leftarrow 0$;
    $maxLabel \leftarrow 0$;
    for $i \leftarrow 1$ to $n$ do
        $cov \leftarrow 1$;
        for $j \leftarrow i + 1$ to $n$ do
            if $x[j] > x[i] + k$ then break;
            $cov \leftarrow cov + 1$;
        if $cov \geq maxCov$ then
            $maxCov \leftarrow cov$;
            $maxLabel \leftarrow i$;
    for $j \leftarrow maxLabel + maxCov$ to $n$ do
        $x[j - maxCov] \leftarrow x[j]$;
    $n \leftarrow n - maxCov$;
    $N \leftarrow N + 1$;
return $N$;
```

To reduce the time complexity on moving elements, the right most end of interval with the same cover number is selected on each loop.

The worst case comes from every range could only cover one elements at a time. Then the time complexity comes to:

$$T(n) = O(n \log n) + \sum_{i=1}^{n} 6i = O(n \log n) + 3n(n + 1) = O(n^2)$$

where one next element is scanned for each investigating element, and only one element is removed from the set for each loop.

Or there is an another *sub-worst case*. The first element cover will contain all the elements in the set, but the scanning process has to go through every element. The time complexity follows:

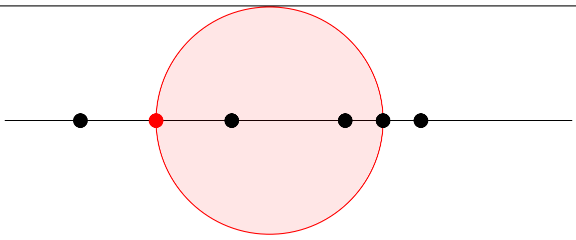
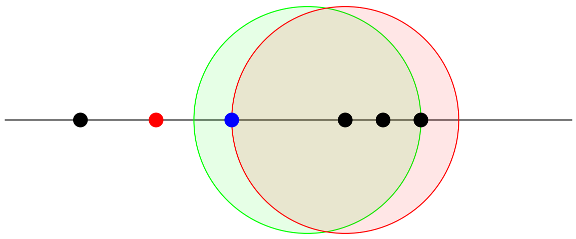
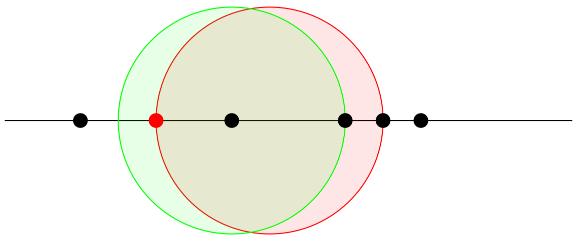
$$T(n) = O(n \log n) + \sum_{i=1}^{n} i = O(n^2)$$

which is the same level of quantity on the time complexity.

(b) **Proof. Proof by contradiction.**

Lemma 1 (Start from an element)**.** *It is optimal to select the intervals that start from an element.*

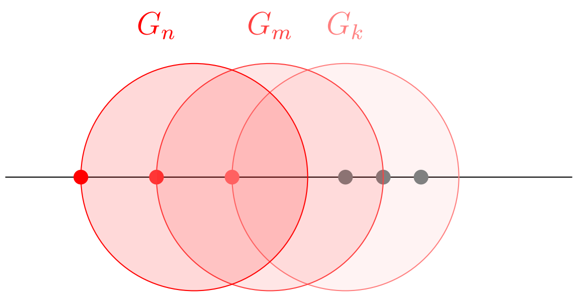
Proof of Lemma 1.

Scenario	**Visual**	**Explaination**
middle		Interval starts from an element.
right shift		If the interval right shift a little bit, the amount of covered elements increases, then the increased elements could also be covered by the interval starting from the next element (blue).
left shift		If the interval left shift a little bit and not touching the previous element, then the amount of covered elements will not exceed the middle scenario, because the lengths of the intervals are the same.

Thus, it is always optimal to select those intervals from an element. □

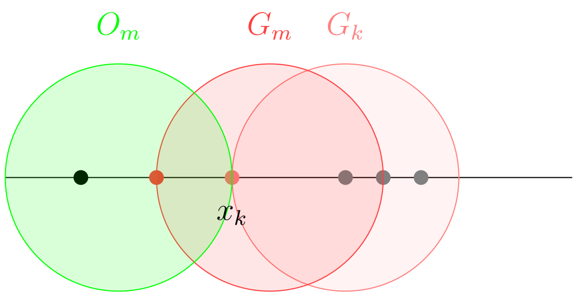
Lemma 2 (At most one overlap on one side)**.** *The intervals selected by the greedy algorithm could only have at most one overlapped area on each side.*

Proof of Lemma 2. Prove by contradiction. If the intervals G_k, G_m, G_n are selected by the greedy algorithm in selection order $(k < m < n)$ and G_m, G_n overlapped with G_k on the same side, they have to be **both on the left side**. Because the right end of the G_k are already covered and no starting elements could be selected to form an inteval for overlapping.



However, as is shown in the diagram, If G_m is replaced by G_n at the stage m, the interval could **cover more elements** because the overlapping area **has already been covered** by G_k, which makes no effect on the number of covering elements. Then, this contradicts the definition of the greedy algorithm, so there could only be at most one overlapping area on each side. □

Then, to construct the optimized solution, the overlapping area has to be reduced comparing to the greedy solution.



Now, consider the overlapped two greedy intervals $G_k, G_m (k < m)$. To reduce the overlapping, construct the optimized interval O_m end at x_k where G_k started and replace G_m by O_m. However, O_m will cover more elements than G_m as left shift of the interval will introduce more covered elements and the right overlapping side is the same for no improvement, **which conflicts the choice of greedy algorithm** at the stage m. So no overlapping could be improved and a better optimized interval does not exist. The greedy solution is the optimal solution. □

(c) `Code-SetCover.cpp`

```cpp
#include <iostream>

using namespace std;

void quickSort(int s[], int l, int r)
{
    if (l< r)
    {
        int i = l, j = r, x = s[l];
        while (i < j)
        {
            while(i < j && s[j]>= x)
                j--;
            if(i < j)
                s[i++] = s[j];
            while(i < j && s[i]< x)
                i++;
            if(i < j)
                s[j--] = s[i];
        }
        s[i] = x;
        quickSort(s, l, i - 1);
        quickSort(s, i + 1, r);
    }
}

int Greedy(int x[], int k, int n)
{
    /*
    Please write your Greedy function here.
    If you want to use sorting, please use the quickSort function above.
    */
    quickSort(x,0,n);

    int num_interval = 0;

    while(n){
        int maxcov = 0;
        int maxlabel = 0;
        for(int i = 0; i < n ; ++i){
            int cov = 1;
            for(int j = i+1; x[j]<=x[i]+k && j<n; ++j)
                ++cov;
            if (cov >= maxcov){
                maxcov = cov;
                maxlabel = i;
            }
        }
        for(int j = maxlabel + maxcov; j < n; ++j)
            x[j - maxcov] = x[j];
        n -= maxcov;
        ++num_interval;
    }

    return num_interval;
}

int main()
{
    //x is the point set P with n=7 nodes in total, and the length of intervals is k=3.
    int x[7]={1,2,3,4,5,6,-2};
    int k=3;
    int n=sizeof(x) / sizeof(x[0]);
    int num_interval=Greedy(x,k,n);
    cout << num_interval << endl;
    return 0;
}
```

The output is:

```
3
```

□

Remark: You need to include your .pdf and .tex files in your uploaded .rar or .zip file.