In [7]:

```
import random
import numpy as np
import copy
import time
import matplotlib.pyplot as plt
import pandas as pd
```

In [8]:

```
random.seed(1)
n=10000 #number of objects
b=200 #number of bins
c=50

#Generate random locations
vj=random.choices(range(10, 100),k=n)
wj=random.choices(range(5, 20),k=n)
```

Solution Encoding

The solution is represented with an n by b matrix where n is the number of items and b is the number of containers. Entry (i,j) represents whether item i is assigned to container j with binary encoding.

The code below randomly generates a solution that might not be feasible. Each item has 1/15 chance to be put in the containers with equal probability.

In [9]:

```
#input: n-number of items; b-number of containers
#output: result-a table of arrangement that might not be feasible
def generate_solution(n,b):
    result=np.zeros((n,b))
    for i in range(n):
        #whether to put the item in a knapsack
        a=random.randint(0,14)
        if a==1:
            #decide to put in which knapsack
            j=random.randint(0,b-1)
            result[i,j]=1
    return result
```

The code below fixes an infeasible solution by randomly removing items from the overweighted bags.

In [10]:

```
#input: x-a table of arrangement that might not be feasible c-capacity
#output: x-a table of a feasible arrangement
def fix(x,c):
    #check whether the result is valid
    n,b=x.shape
    valid=True
    overweight={}
    for i in range(b):
        weight=0
        for j in range(n):
            weight+=wj[j]*x[j,i]
        if weight>c:
            valid=False
            overweight[i]=weight

    #randomly remove items until it meets the constraint
    if valid==False:
        for i,w in overweight.items():
            in_bag=[]
            for j in range(n):
                if x[j,i]==1:
                    in_bag.append(j)
            while w>c:
                a=random.randint(0,len(in_bag)-1)
                x[in_bag[a],i]=0
                w-=wj[in_bag[a]]
                in_bag.pop(a)
    return x
```

The code below is used to convert a numpy result table to a more readable one.

In [11]:

```
def final_table(x):
        container=list(range(1,b+1))
        container.append('Total')
        package=[]
        for i in range(b):
            package.append([])
            for j in range(n):
                if x[j,i]==1:
                    package[i].append(j+1)
        package.append('')

        profits=[]
        weights=[]
        for i in range(b):
            profit=0
            weight=0
            for j in package[i]:
                profit+=vj[j-1]
                weight+=wj[j-1]
            profits.append(profit)
            weights.append(weight)
        profits.append(sum(profits))
        weights.append('')
        data={'Container':container,'Packages':package,'Profit':profits,'Weight':weights}
        return pd.DataFrame(data)
```

Mutation

For each item, it has a 0.1% chance of mutation. After the mutation, it will either be placed in another bag, or just removed from the bag with equal probability.

In [12]:

```
def mutation(x):
    n,b=x.shape
    for i in range(n):
        #randomly choose a number among 1 to 1000, the number has 0.1% chance to be 100
        m=random.randint(1,1000)
        if m==100:
            for j in range(b):
                if x[i,j]==1:
                    x[i,j]=0
            #randomly choose a container to put the item in, if k=b, then just remove the item from
            k=random.randint(0,b)
            if k!=b:
                x[i,k]=1
    return x
```

Crossover

The crossover here is uniform crossover with p=0.5

In [13]:

```
#input: x,y-two feasible arrangements act as parents
#output: result-the offspring
def crossover(x,y):
    n,b=x.shape
    result=np.zeros((n,b))
    for i in range(n):
        a=random.randint(0,1)
        if a==0:
            for j in range(b):
                result[i,j]=x[i,j]
        else:
            for j in range(b):
                result[i,j]=y[i,j]
    return result
```

Fitness

Fitness in this case is just the total amount of profit.

In [14]:

```
#input: x-a table of a feasible arrangement
#output: profit-the fitness/total profit of this arrangement
def fitness(x):
    n,b=x.shape
    profit=0
    for i in range(n):
        for j in range(b):
            profit+=vj[i]*x[i,j]
    return profit
```

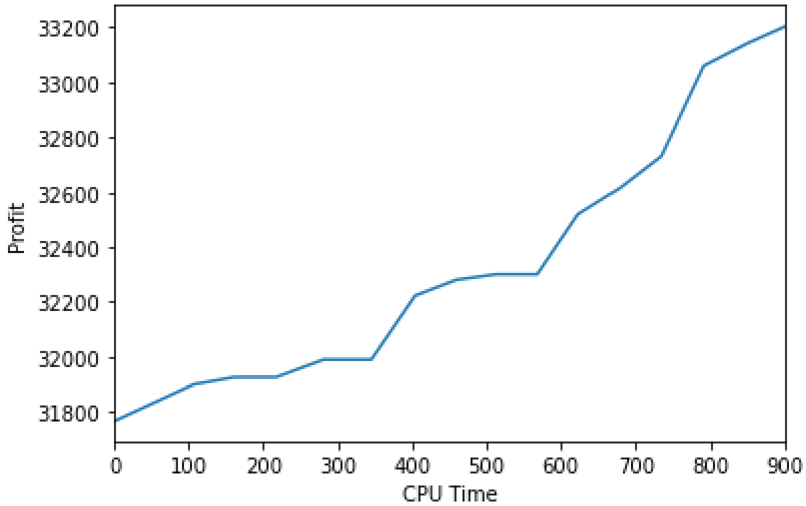
The initial population are uniformly randomly generated. The population size is 10. Mutation with probability of 1% is used as inertial operator. Uniform crossover with personal best and global best are used as cognitive operator and social operator.

In [15]:

```
start=time.time()
#generate 10 initial solutions
population=[]
pbest_list=[]
pbest_fitness=[]
for i in range(10):
    a=generate_solution(n,b)
    d=fix(a,c)
    population.append(d)
    pbest_list.append(d)
    pbest_fitness.append(fitness(d))

gbest=population[0]
gbest_fitness=fitness(population[0])
for i in population:
    if fitness(i)>gbest_fitness:
        gbest=copy.deepcopy(i)
        gbest_fitness=fitness(i)

end=time.time()
time_list=[0]
value_list=[gbest_fitness]
while end-start<60*15:
    #Calculate the new state for each solution and update pbest
    for i in range(len(population)):
        j=mutation(population[i]) #inertia
        k=crossover(j,pbest_list[i]) #cognitive
        l=crossover(k,gbest) #social
        m=fix(l,c)
        population[i]=copy.deepcopy(m)
        if fitness(m)>pbest_fitness[i]:
            pbest_list[i]=copy.deepcopy(m)
            pbest_fitness[i]=fitness(m)
    #Update gbest
    for i in range(len(population)):
        if pbest_fitness[i]>gbest_fitness:
            gbest=copy.deepcopy(pbest_list[i])
            gbest_fitness=pbest_fitness[i]

    end=time.time()
    time_list.append(end-start)
    value_list.append(gbest_fitness)

plt.xlabel('CPU Time')
plt.ylabel('Profit')
plt.xlim((0,60*15))
plt.plot(time_list,value_list)
```

Out[15]:

```
[<matplotlib.lines.Line2D at 0x1a62fbdaa90>]
```



The terrible result is mainly because the algorithm only completed 16 iterations within 15 minutes. It is quite hard to deal with 20 200x10000 matrices.

In [17]:

```
print(len(value_list))
print(time_list)
```

```
16
[0, 106.86642265319824, 160.41129612922668, 217.04593920707703, 280.87535667419434,
345.09871530532837, 403.84272360801697, 459.48106360435486, 513.0698373317719, 568.0
638380050659, 622.4624578952789, 678.8926446437836, 734.2786245346069, 791.121713876
7242, 849.7500245571136, 905.0592386722565]
```

As you can see, it takes about a minute to complete 1 iteration.