Assignment 1

Instructions:

- This is an individual assignment. You are not allowed to discuss the problems with other students.
- Part of this assignment will be autograded by gradescope. You can use it as immediate feedback to improve your answers. You can resubmit as many times as you want.
- All your solution, code, analysis, graphs, explanations should be done in this same notebook.
- Please make sure to execute all the cells before you submit the notebook to the gradescope. You will not get points for the plots if they are not generated already.
- If you have questions regarding the assignment, you can ask for clarifications in Piazza. You should use the corresponding tag for this assignment.

When Submitting to GradeScope: Be sure to 1) Submit a `.ipynb` notebook to the `Assignment 1 - Code` section on Gradescope. 2) Submit a `pdf` version of the notebook to the `Assignment 1 - Report` entry and tag the answers.

Note: You can choose to submit responses in either English or French.

Before starting the assignment, make sure that you have downloaded all the tests related for the assignment and put them in the appropriate locations. If you run the next cell, we will set this all up automatically for you in a dataset called public, which will contain both the data and tests you use.

This assignment has only one question. In this question, you will learn:

1. To understand how to formalize a dose finding study as a multi-arm bandit problem.
2. To implement ϵ**-greedy**, **UCB**, **Boltzmann**, and **Gradient bandit** algorithms.
3. Understand the role of different hyper-parameters.

In [1]:
```
!pip install -q otter-grader
!git clone https://github.com/chandar-lab/INF8250ae-assignments-2023.git public
```

```
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 158.0/158.0 kB 6.0 MB/s eta 0:00:
00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 115.3/115.3 kB 14.4 MB/s eta 0:00:
00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 157.9/157.9 kB 20.5 MB/s eta 0:00:
00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 307.2/307.2 kB 34.2 MB/s eta 0:00:
00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 100.2/100.2 kB 12.8 MB/s eta 0:00:
00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.6/1.6 MB 57.7 MB/s eta 0:00:00
Cloning into 'public'...
remote: Enumerating objects: 45, done.
remote: Counting objects: 100% (45/45), done.
remote: Compressing objects: 100% (23/23), done.
remote: Total 45 (delta 17), reused 37 (delta 12), pack-reused 0
Receiving objects: 100% (45/45), 4.96 KiB | 2.48 MiB/s, done.
Resolving deltas: 100% (17/17), done.
```

In [2]:

```python
import otter
grader = otter.Notebook(colab=True, tests_dir='./public/a1/tests')
```

In [3]:

```python
import numpy as np
from random import choice
from scipy.stats import bernoulli
from typing import Sequence, Tuple
import matplotlib.pyplot as plt
%matplotlib inline
np.random.seed(8953)
import warnings
warnings.filterwarnings('ignore')
```

Q1: Dose Finding Study (90 points)

In the context of clinical trials, Phase I trials are the first stage of testing in human subjects. Their goal is to evaluate the safety (and feasibility) of the treatment and identify its side effects. The aim of a phase I dose-finding study is to determine the most appropriate dose level that should be used in further phases of the clinical trials. Traditionally, the focus is on determining the highest dose with acceptable toxicity called the Maximum Tolerated Dose (MTD).

A dose-finding study involves a number K of dose levels that have been chosen by physicians based on preliminary experiments (K is usually a number between 3 and 10). Denoting by p_k the (unknown) toxicity probability of dose k, the Maximum Tolerated Dose (MTD) is defined as the dose with a toxicity probability closest to a target:

$$k^* \in \underset{k \in \{1,...,K\}}{\text{argmin}} |\theta - p_k| \tag{1}$$

where θ is the pre-specified targeted toxicity probability (typically between 0.2 and 0.35). A MTD identification algorithm proceeds sequentially: at round t a dose $D_t \in \{1, \ldots, K\}$ is selected and administered to a patient for whom a toxicity response is observed. A binary outcome X_t is revealed where $X_t = 1$ indicates that a harmful side-effect occurred and

$X_t = 0$ indicates than no harmful side-effect occurred. We assume that X_t is drawn from a Bernoulli distribution with mean p_{D_t} and is independent from previous observations.

Hint: In this example, the reward definition is a bit different from the usual case. We would like to take the arm with minimum $|\theta - \hat{p}_k|$ where $\hat{p}_k$ is the estimated toxicity probability.

Q1.a: Define your Bandit class (5 points):

Most of the class has been written. Complete the pull method in such a way that:

1. Update both `num_dose_selected` and `num_toxic` arrays,

2. Compute and return the reward $-|\theta - \hat{p}_k|$ where $\hat{p}_k$ is the estimated toxicity probability of arm k.

In [4]:
```
class Bandit(object):

  def __init__(self,
               n_arm: int = 2,
               n_pulls: int = 2000,
               actual_toxicity_prob: list = [0.4, 0.6],
               theta: float = 0.3,
               ):
    self.n_arm = n_arm
    self.n_pulls = n_pulls
    self.actual_toxicity_prob = actual_toxicity_prob
    self.theta = theta
    self.init_bandit()

  def init_bandit(self):
    """
        Initialize the bandit
    """
    self.num_dose_selected = np.array([0]*self.n_arm) # number of times a dose
    self.num_toxic = np.array([0]*self.n_arm) # number of times a does found tc

  def pull(self, a_idx: int):
    """
    .inputs:
      a_idx: Index of action.
    .outputs:
      rew: reward value.
    """
    assert a_idx < self.n_arm, "invalid action index"
    # ----------------------------------------------
    self.num_dose_selected[a_idx] += 1
    is_toxic = int(bernoulli.rvs(self.actual_toxicity_prob[a_idx]))
    self.num_toxic[a_idx] += is_toxic
    # estimate the probabilties pk with MLE
    rew = -abs(self.theta - (float(self.num_toxic[a_idx])/float(self.num_dose_s
    # ----------------------------------------------
    return rew
```

In [5]:
```
grader.check("q1a")
```

Out[5]:

q1.a passed! 🍀

Dose finding study with three doses

Let's define a dose finding study with three doses ($K = 3$) where you need to choose from with `actual_toxicity_prob=[0.1, 0.35, 0.8]` and targeted toxicity probability is $\theta = 0.3$.

In [6]:
```
#@title Problem definition
bandit = Bandit(n_arm=3, n_pulls=2000, actual_toxicity_prob=[0.1, 0.35, 0.8], t
```

Q1.b: ϵ-greedy for k-armed bandit and Optimistic initial values (25 points)

Q1.b1: ϵ-greedy algorithm implementation (5 points)

Implement the ϵ-greedy method.

In [7]:
```
def eps_greedy(
    bandit: Bandit,
    eps: float,
    init_q: float = .0
    ) -> Tuple[list, list, list]:
  """
  .inputs:
    bandit: A bandit problem, instantiated from the above class.
    eps: The epsilon value.
    init_q: Initial estimation of each arm's value.
  .outputs:
    rew_record: The record of rewards at each timestep.
    avg_ret_record: The average of rewards up to step t, where t goes from 0 to
    we define `ret_T` = \sum^T_{t=0}{r_t}, `avg_ret_record` = ret_T / (1+T).
    tot_reg_record: The  regret up to step t, where t goes from 0 to n_pulls.
    opt_action_perc_record: Percentage of optimal arm selected.
  """
  # initialize q values
  q = np.array([init_q]*bandit.n_arm, dtype=float)

  ret = .0
  rew_record = []
  avg_ret_record = []
  tot_reg_record = []
  opt_action_perc_record = []

  true_action_rewards = -np.abs(bandit.theta-np.array(bandit.actual_toxicity_pr
  optimal_reward = np.max(true_action_rewards)
  optimal_action = np.argmax(true_action_rewards)

  for t in range(bandit.n_pulls):
    # -----------------------------------------------
    sample = np.random.rand()
    if sample > eps:
      # [Exploit] greedy action while breaking ties randomly.
```

```
            max_value = np.max(q)
            max_indicies = np.where(q == max_value)[0]
            chosen_arm = np.random.choice(max_indicies)

        else:
            # [Explore] choose a random arm
            chosen_arm = np.random.choice(bandit.n_arm)
        chosen_arm = int(chosen_arm)
        reward = bandit.pull(chosen_arm)
        rew_record.append(reward)

        # epsilon-greedy update rule
        #   num_dose_selected will be automatically updated when the arm is pulled.

        q_a = q[chosen_arm]
        n_a = bandit.num_dose_selected[chosen_arm]
        q_a = q_a + (1./n_a) * (reward - q_a)
        q[chosen_arm] = q_a

        opt_action_perc_record.append(100*bandit.num_dose_selected[optimal_action]/

    returns = np.cumsum(rew_record)
    denoms = np.arange(len(returns))
    denoms += 1
    avg_ret_record = returns/denoms

    cumulative_optimal_rewards = denoms * optimal_reward
    tot_reg_record = cumulative_optimal_rewards - returns

    tot_reg_record.tolist()
    avg_ret_record.tolist()
    tot_reg_record.tolist()
      # ----------------------------------------------

    return rew_record, avg_ret_record, tot_reg_record, opt_action_perc_record
```

In [8]:
```
grader.check("q1b1")
```

Out[8]:

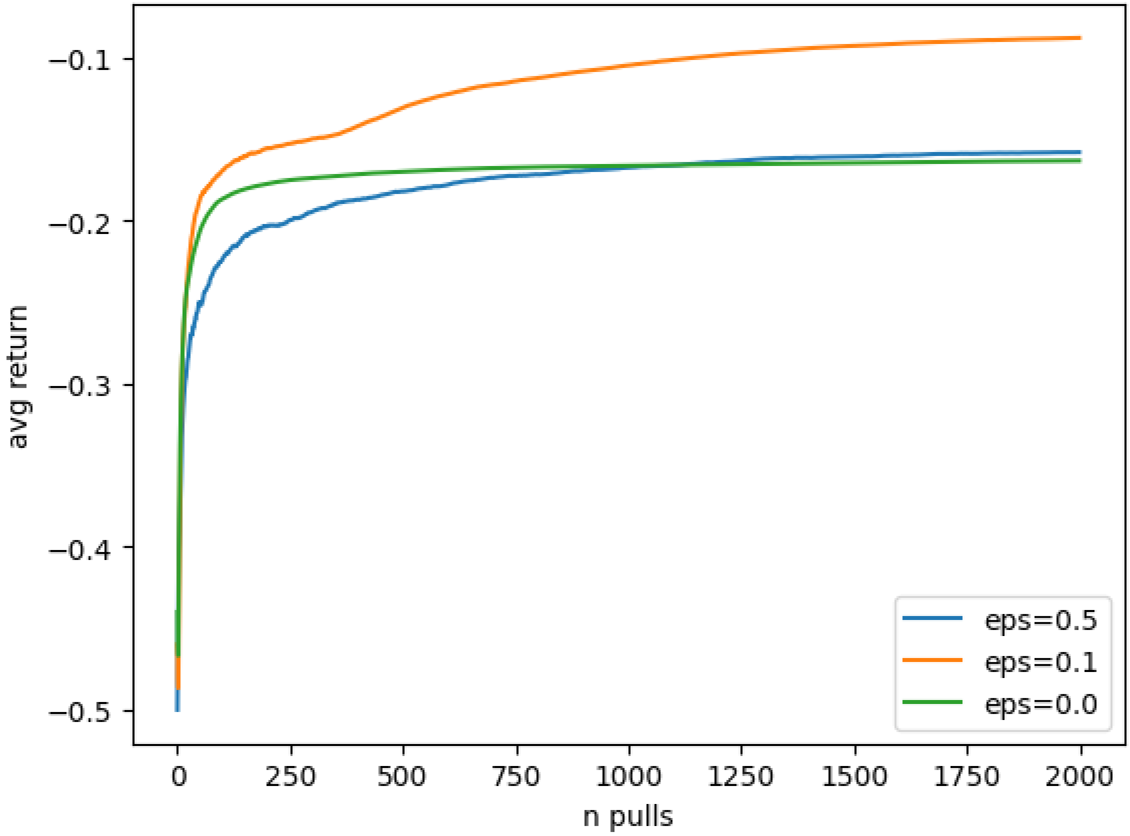
q1.b1 passed! ✨

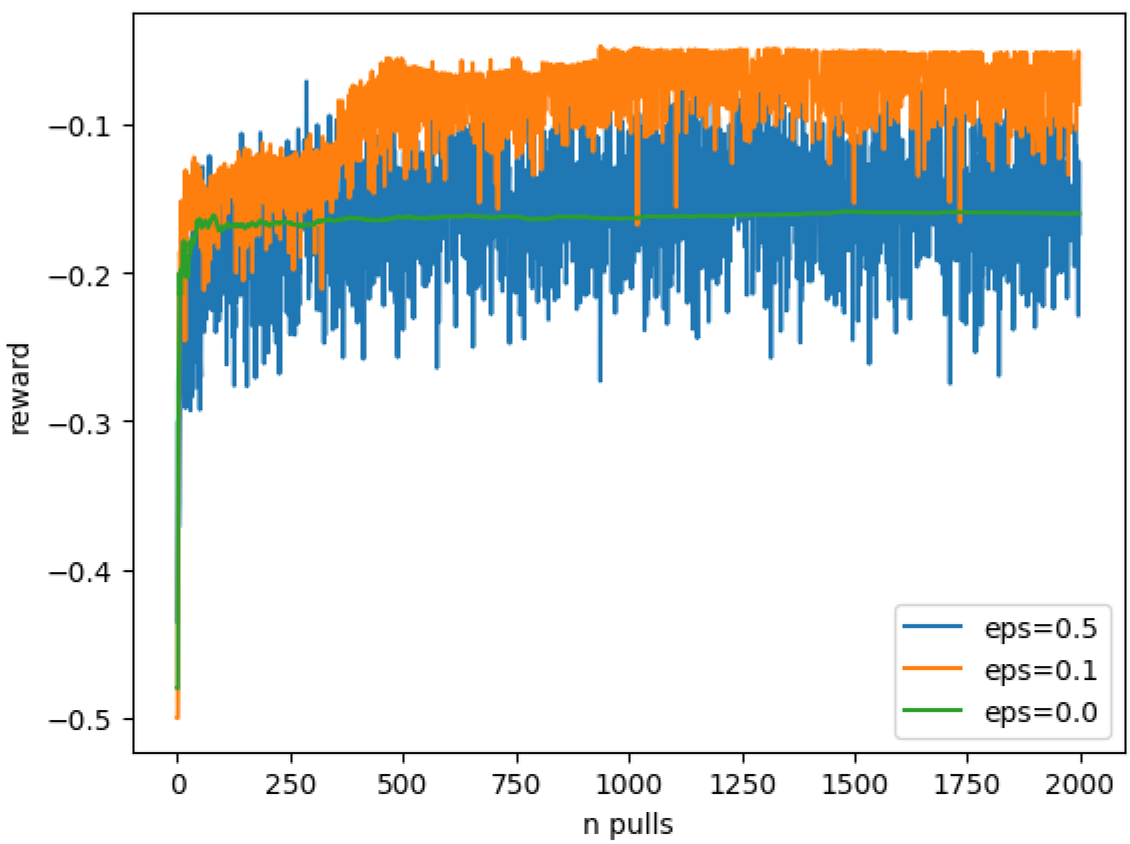
Q1.b2: Plotting the results (5 points)

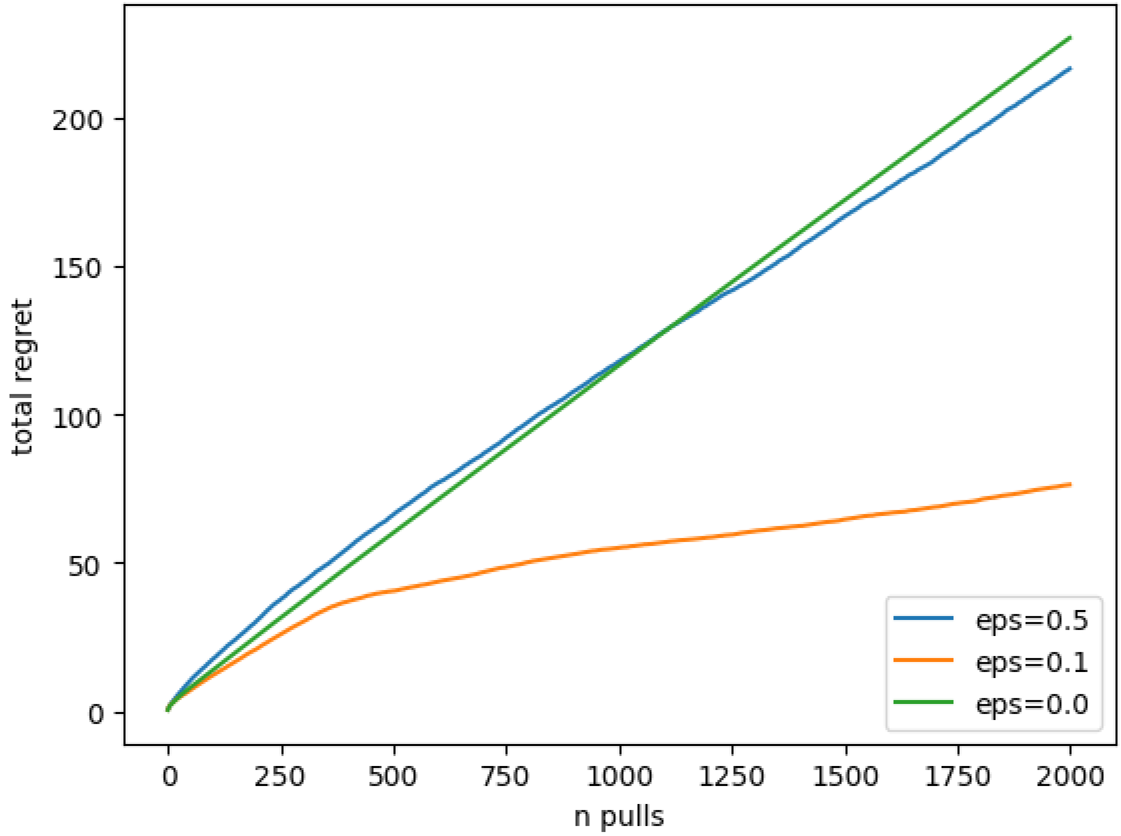
Use the driver code provided to plot: (1) The average return, (2) The reward, (3) the total regret, and (4) the percentage of optimal action across the N=20 runs as a function of the number of pulls (2000 pulls for each run) for all three ϵ values of 0.5, 0.1, and 0.

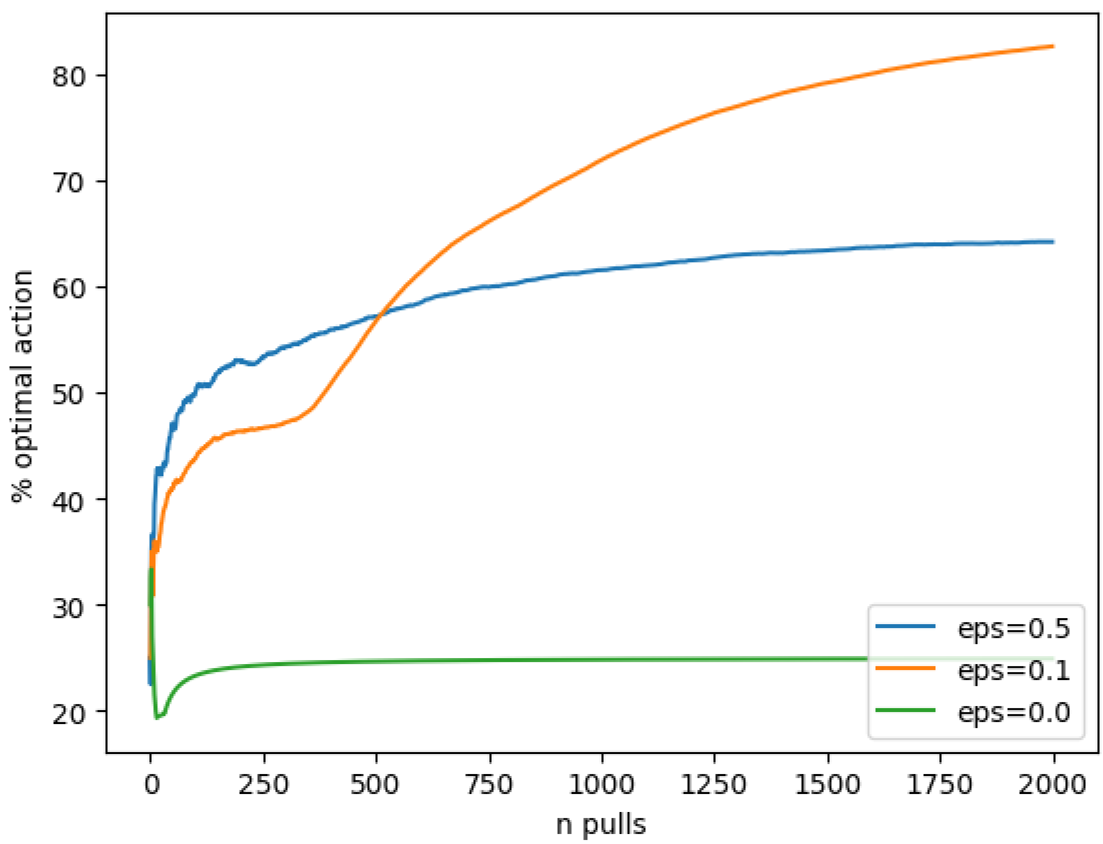
In [10]:
```
import time
plt.figure(0)
plt.xlabel("n pulls")
plt.ylabel("avg return")
plt.figure(1)
plt.xlabel("n pulls")
plt.ylabel("reward")
plt.figure(2)
plt.xlabel("n pulls")
plt.ylabel("total regret")
```

```
plt.figure(3)
plt.xlabel("n pulls")
plt.ylabel("% optimal action")

N = 20
tot_reg_rec_best = 1e8

for eps in [0.5, 0.1, .0]:
  rew_rec = np.zeros(bandit.n_pulls)
  avg_ret_rec = np.zeros(bandit.n_pulls)
  tot_reg_rec = np.zeros(bandit.n_pulls)
  opt_act_rec = np.zeros(bandit.n_pulls)
  start_time = time.time()
  for n in range(N):
    bandit.init_bandit()
    rew_rec_n, avg_ret_rec_n, tot_reg_rec_n, opt_act_rec_n = eps_greedy(bandit,
    rew_rec += np.array(rew_rec_n)
    avg_ret_rec += np.array(avg_ret_rec_n)
    tot_reg_rec += np.array(tot_reg_rec_n)
    opt_act_rec += np.array(opt_act_rec_n)

  end_time = time.time()
  # print(f"time per run: {end_time - start_time}/N")
  # take the mean
  rew_rec /= N
  avg_ret_rec /= N
  tot_reg_rec /= N
  opt_act_rec /= N

  plt.figure(0)
  plt.plot(avg_ret_rec, label="eps={}".format(eps))
  plt.legend(loc="lower right")

  plt.figure(1)
  plt.plot(rew_rec[1:], label="eps={}".format(eps))
  plt.legend(loc="lower right")

  plt.figure(2)
  plt.plot(tot_reg_rec, label="eps={}".format(eps))
  plt.legend(loc="lower right")

  plt.figure(3)
  plt.plot(opt_act_rec, label="eps={}".format(eps))
  plt.legend(loc="lower right")


  if tot_reg_rec[-1] < tot_reg_rec_best:
        ep_greedy_dict = {
        'opt_act':opt_act_rec,
        'regret_list':tot_reg_rec,}
        tot_reg_rec_best = tot_reg_rec[-1]
```


avg return
−0.1
−0.2
−0.3
−0.4
−0.5
0
250
500
750
1000
1250
1500
1750
2000
n pulls
eps=0.5
eps=0.1
eps=0.0


reward
−0.1
−0.2
−0.3
−0.4
−0.5
0
250
500
750
1000
1250
1500
1750
2000
n pulls
eps=0.5
eps=0.1
eps=0.0





Q1.b3: Analysis (5 points)

Explain the results from the perspective of exploration and how different ϵ values affect the results.

Asymptotic behavior:

- It is clear from the figures that using epsilon = 0.1 yields better results in the long run, when ϵ = 0.1 we're approaching %optimal action that is closer to 85%, and yields less regret compared to Other values. Using ϵ = 0.5 yields more regret than ϵ = 0.1 and less than the greedy exploitation with ϵ = 0.0, which is the worst of them.

Initial behavior

- The greedy exploitation can do slightly better in the beginning (this is expected since it is greedy) in terms of average return, but quickly the exploration with ϵ = 0.5 will catch up to it and surpass it.

Conclusion

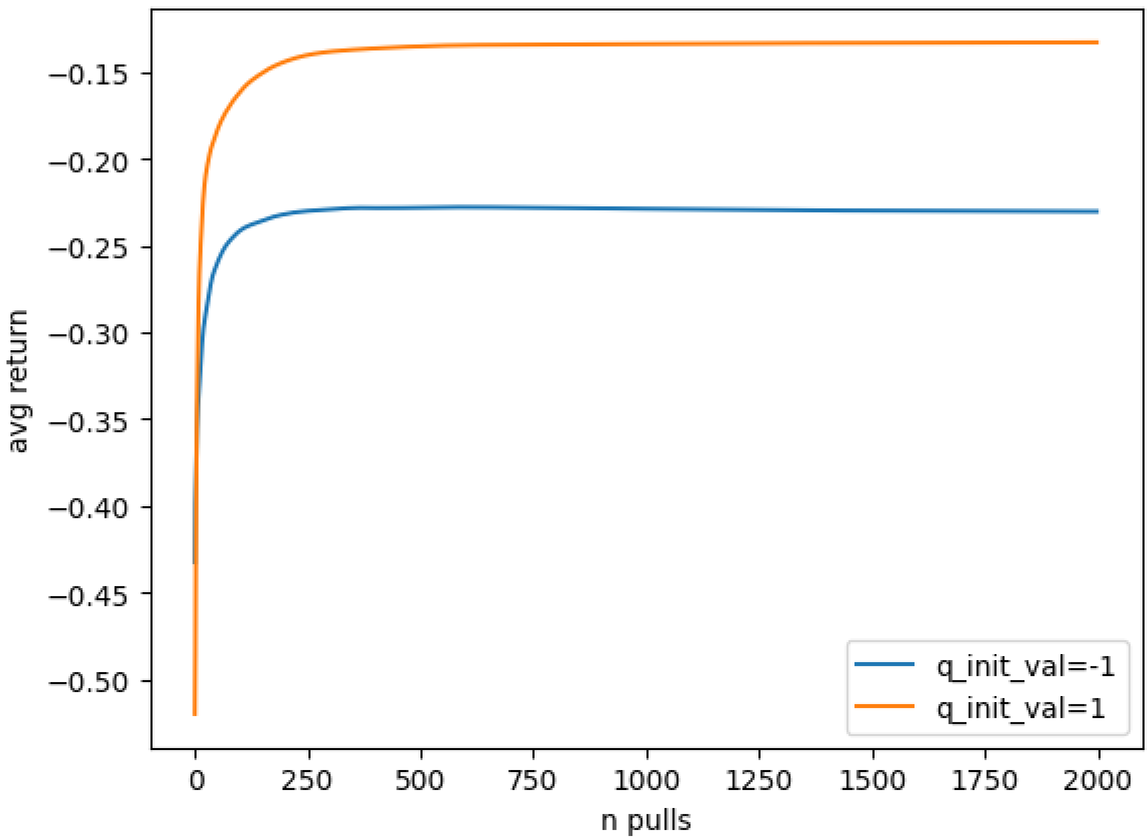
- Having a relatively small expolration precentage ϵ (10%) is the best thing to do. Having a very large exploration percentage will slow down reaching the best outcome and no exploration is the worst.

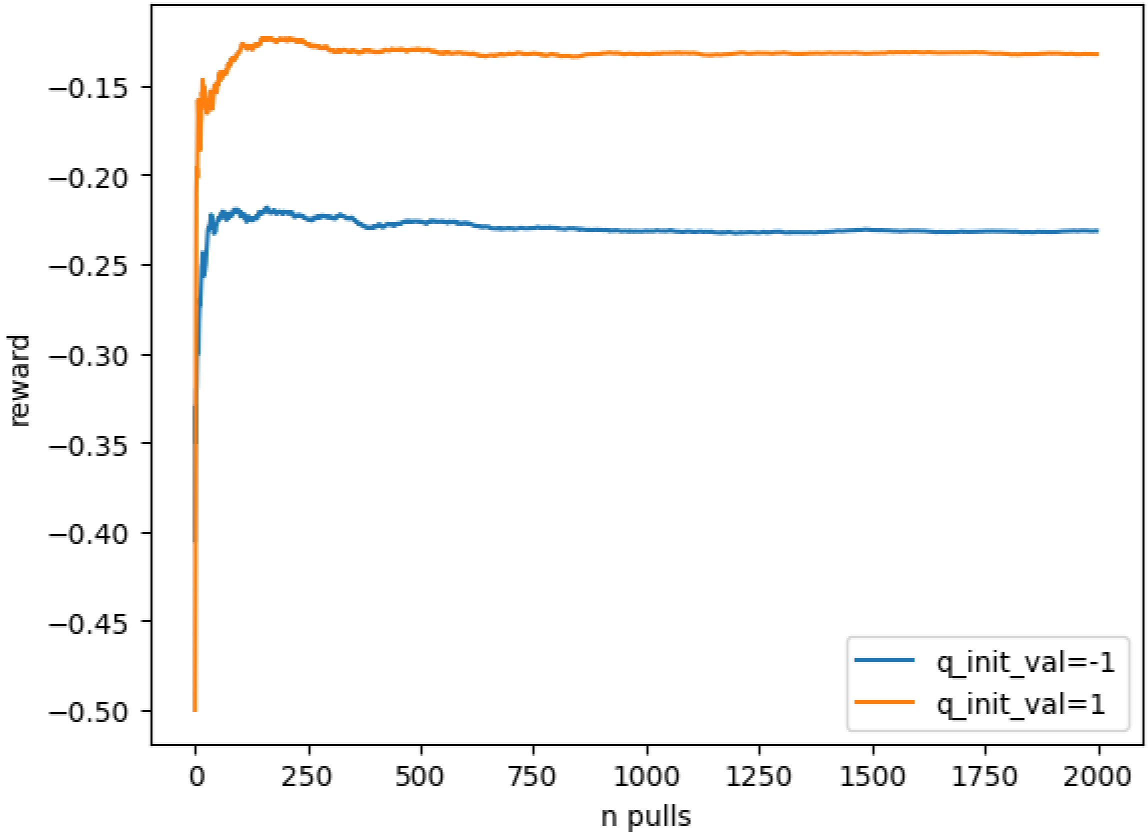
Q1.b4: Optimistic Initial Values (5 points)

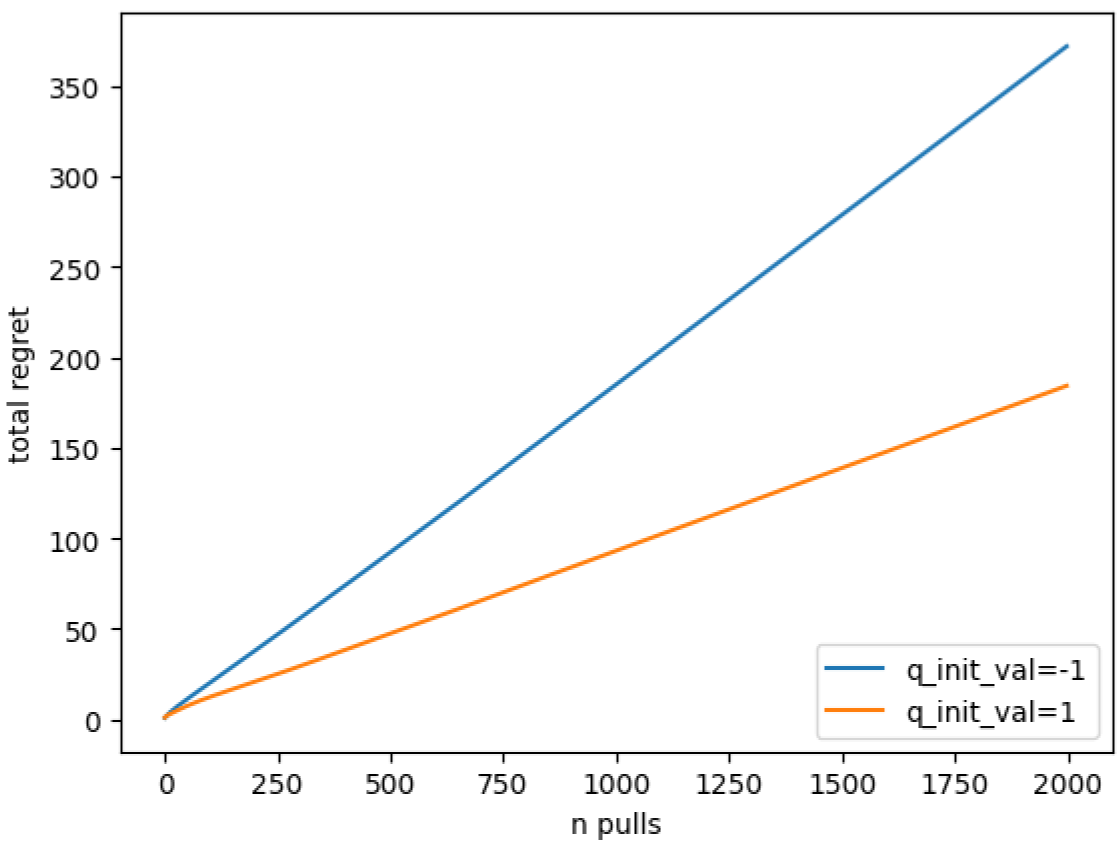
We want to run the optimistic initial value method on the same problem described above for the initial q values of -1 and +1 for all arms. Compare its performance, measured by the average reward across N=20 runs as a function of the number of pulls, with the non-optimistic setting with initial q values of 0 for all arms. For both optimistic and non-optimistic settings, ϵ=0.

In [11]:
```
plt.figure(4)
plt.xlabel("n pulls")
plt.ylabel("avg return")

plt.figure(5)
plt.xlabel("n pulls")
plt.ylabel("reward")

plt.figure(6)
plt.xlabel("n pulls")
plt.ylabel("total regret")

N = 20
for init_q in [-1, 1]:
  rew_rec = np.zeros(bandit.n_pulls)
  avg_ret_rec = np.zeros(bandit.n_pulls)
  for n in range(N):
    bandit.init_bandit()
    rew_rec_n, avg_ret_rec_n, tot_reg_rec_n, opt_act_rec_n = eps_greedy(bandit,

    rew_rec += np.array(rew_rec_n)
    avg_ret_rec += np.array(avg_ret_rec_n)
```

```
    tot_reg_rec += np.array(tot_reg_rec_n)

  avg_ret_rec /= N
  rew_rec /= N
  tot_reg_rec /= N
  plt.figure(4)
  plt.plot(avg_ret_rec[1:], label="q_init_val={}".format(init_q))
  plt.legend(loc="lower right")

  plt.figure(5)
  plt.plot(rew_rec[1:], label="q_init_val={}".format(init_q))
  plt.legend(loc="lower right")

  plt.figure(6)
  plt.plot(tot_reg_rec[1:], label="q_init_val={}".format(init_q))
  plt.legend(loc="lower right")
```







Q1.b5: Analysis (5 points)

Explain how initial q values affect the exploration and the performance.

- Having larger initial Q value *(optimistic intitalization)* encourages exploration and asymptotically it yields higher average return, higher reward, and less regret.

Q1.c: Upper-Confidence-Bound action selection (15 points)

Q1.c1: UCB algorithm implementation (5 points)

Implement the UCB algorithm on the same MAB problem as above.

In [12]:
```
def ucb(
    bandit: Bandit,
    c: float,
    init_q: float = .0
    ) -> Tuple[list, list, list]:
  """
  .inputs:
    bandit: A bandit problem, instantiated from the above class.
    c: The additional term coefficient.
    init_q: Initial estimation of each arm's value.
  .outputs:
    rew_record: The record of rewards at each timestep.
    avg_ret_record: The average summation of rewards up to step t, where t goes
    we define `ret_T` = \sum^T_{t=0}{r_t}, `avg_ret_record` = ret_T / (1+T).
    tot_reg_record: The  regret up to step t, where t goes from 0 to n_pulls.
    opt_action_perc_record: Percentage of optimal arm selected.
  """
  # init q values (the estimates)
  q = np.array([init_q]*bandit.n_arm, dtype=float)

  ret = .0
  rew_record = []
  avg_ret_record = []
  tot_reg_record = []
  opt_action_perc_record = []

  true_action_rewards = -np.abs(bandit.theta-np.array(bandit.actual_toxicity_pr
  optimal_reward = np.max(true_action_rewards)
  optimal_action = np.argmax(true_action_rewards)

  for t in range(bandit.n_pulls):
    # Assuming to take the first arm always when there is no exploration
    # ---------------------------------------------
    chosen_arm = int(np.argmax(q + c * np.sqrt(np.log(t+1)/np.array(bandit.num_
    reward = bandit.pull(chosen_arm)
    rew_record.append(reward)

    # update rule
    #   num_dose_selected will be automatically updated when the arm is pulled.

    q_a = q[chosen_arm]
    n_a = bandit.num_dose_selected[chosen_arm]
    q_a = q_a + (1./n_a) * (reward - q_a)
    q[chosen_arm] = q_a

    opt_action_perc_record.append(100*bandit.num_dose_selected[optimal_action]/
```

```
  returns = np.cumsum(rew_record)
  denoms = np.arange(len(returns))
  denoms += 1
  avg_ret_record = returns/denoms

  cumulative_optimal_rewards = denoms * optimal_reward
  tot_reg_record = cumulative_optimal_rewards - returns

  tot_reg_record.tolist()
  avg_ret_record.tolist()
  tot_reg_record.tolist()
    # ----------------------------------------------


  return rew_record, avg_ret_record, tot_reg_record, opt_action_perc_record
```

In [13]:

```
grader.check("q1c1")
```

Out[13]:

q1.c1 passed! 🍀

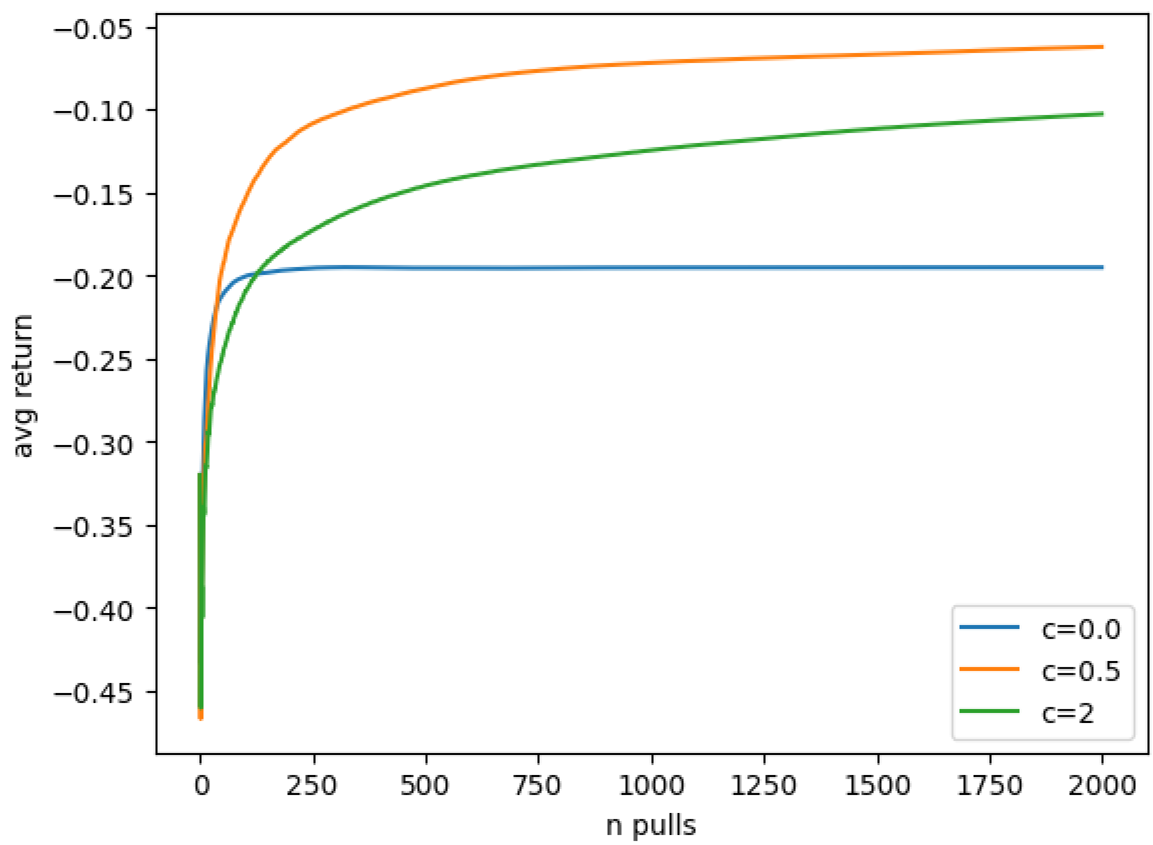
Q1.c2: Plotting the results (5 points)

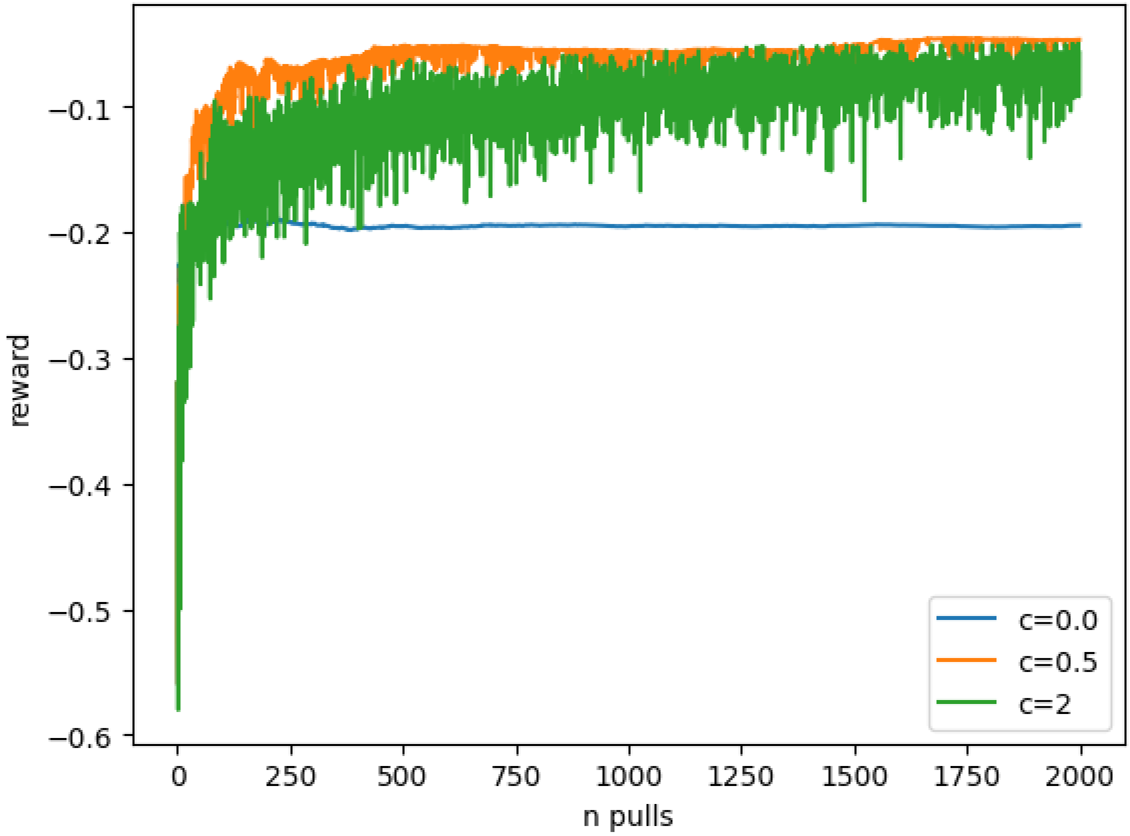
Use the driver code provided to plot: (1) The average return, (2) The reward, (3) the total regret, and (4) the percentage of optimal action across the N=20 runs as a function of the number of pulls (2000 pulls for each run) for three values of c=0, 0.5, and 2.0.

In [14]:

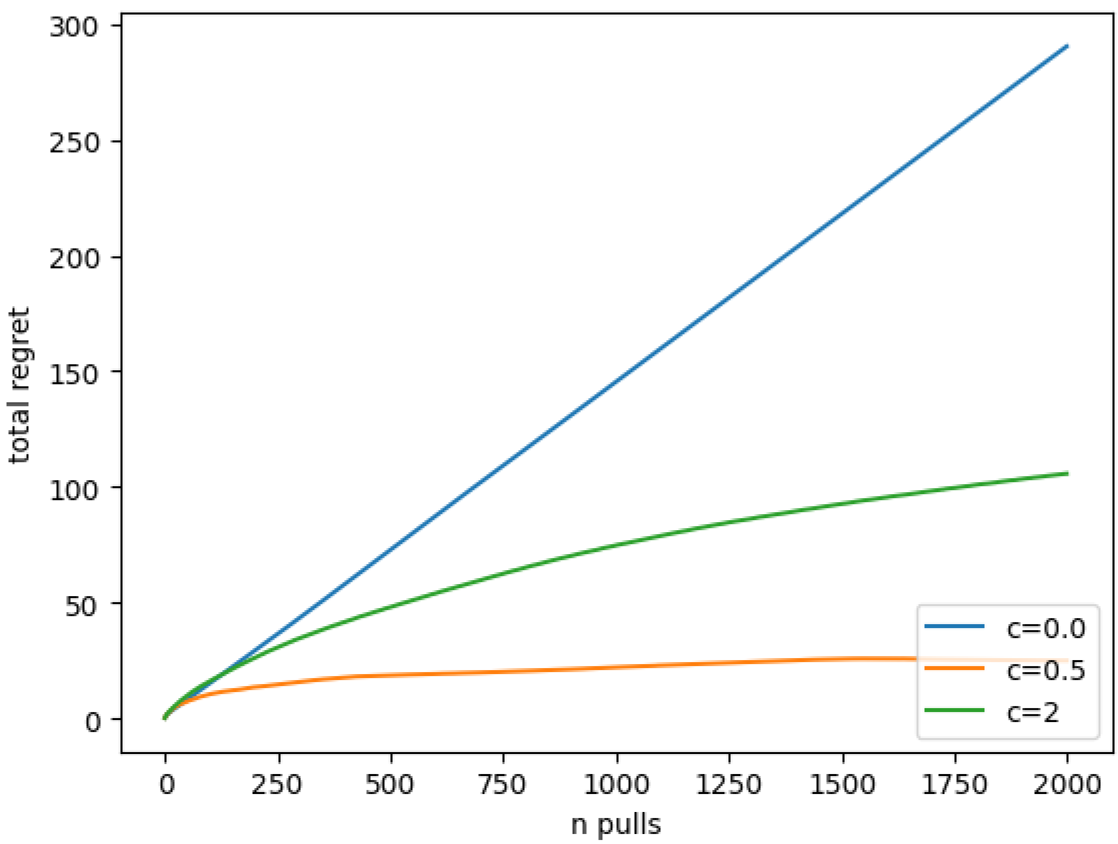
```
plt.figure(7)
plt.xlabel("n pulls")
plt.ylabel("avg return")
plt.figure(8)
plt.xlabel("n pulls")
plt.ylabel("reward")
plt.figure(9)
plt.xlabel("n pulls")
plt.ylabel("total regret")
plt.figure(10)
plt.xlabel("n pulls")
plt.ylabel("% optimal action")

N = 20
tot_reg_rec_best = 1e8
for c in [.0, 0.5, 2]:
  rew_rec = np.zeros(bandit.n_pulls)
  avg_ret_rec = np.zeros(bandit.n_pulls)
  tot_reg_rec = np.zeros(bandit.n_pulls)
  opt_act_rec = np.zeros(bandit.n_pulls)

  for n in range(N):
    bandit.init_bandit()
    rew_rec_n, avg_ret_rec_n, tot_reg_rec_n, opt_act_rec_n = ucb(bandit, c)
    rew_rec += np.array(rew_rec_n)
    avg_ret_rec += np.array(avg_ret_rec_n)
    tot_reg_rec += np.array(tot_reg_rec_n)
    opt_act_rec += np.array(opt_act_rec_n)

  # take the mean
```

```
rew_rec /= N
avg_ret_rec /= N
tot_reg_rec /= N
opt_act_rec /= N

plt.figure(7)
plt.plot(avg_ret_rec, label="c={}".format(c))
plt.legend(loc="lower right")

plt.figure(8)
plt.plot(rew_rec, label="c={}".format(c))
plt.legend(loc="lower right")

plt.figure(9)
plt.plot(tot_reg_rec, label="c={}".format(c))
plt.legend(loc="lower right")

plt.figure(10)
plt.plot(opt_act_rec, label="c={}".format(c))
plt.legend(loc="lower right")

if tot_reg_rec[-1] < tot_reg_rec_best:
      ucb_dict = {
      'opt_act':opt_act_rec,
      'regret_list':tot_reg_rec,}
      tot_reg_rec_best = tot_reg_rec[-1]
```

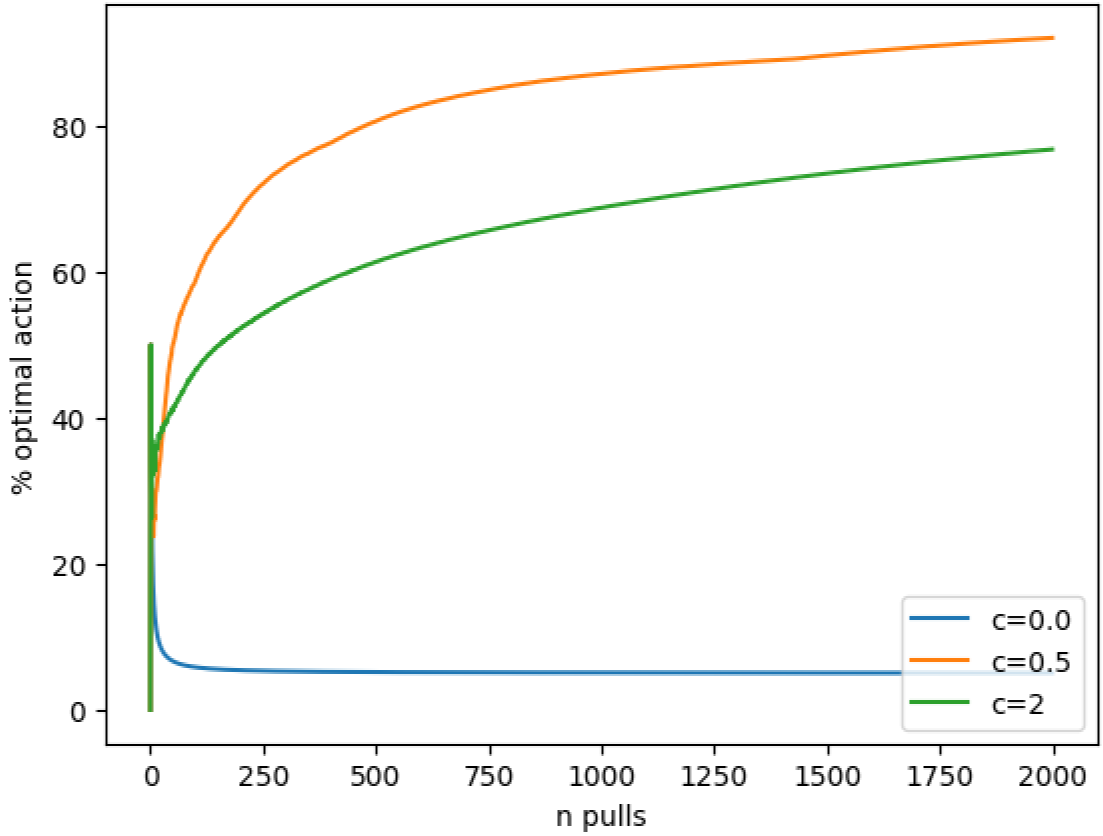




−0.1
−0.2
−0.3
−0.4
−0.5
−0.6
reward
0
250
500
750
1000
1250
1500
1750
2000
n pulls
c=0.0
c=0.5
c=2



300
250
200
150
100
50
0
total regret
0
250
500
750
1000
1250
1500
1750
2000
n pulls
c=0.0
c=0.5
c=2



Q1.c3: Analysis (5 points)

Explain the results from the perspective of exploration and how different c values affect the results.

- The higher the value of c, the more encouragement for exploration we give (more weight for exploration vs exploitiation).
- Clearly from the plots, when c=0.0 we do no exploration, so we're exploiting only which leads to worst results. Having an intermediate c value (0.5) in this case yeilds the best results in tersm of higher return, higher average rewards, higher % of optimal reward and lowest regret. Having a relativly large exploration factor (c=2) slows the algorithm, because we're exploring a lot.

Q1.d: Boltzmann algorithm (20 points)

Q1.d1: Boltzmann policy implementation (5 points)

Implement a Boltzmann policy that gets an array and temprature value (τ) and returns an index sampled from the Boltzmann policy.

In [15]:
```
from math import exp
def boltzmann_policy(x, tau):
    """ Returns softmax probabilities with temperature tau
        Input:  x -- 1-dimensional array
        Output: idx -- chosen index
```

```
    """
    # ----------------------------------------------
    x = np.array(x)/tau
    exponents = np.exp(x)
    probs = exponents/np.sum(exponents, axis=0)
    idx = np.random.choice(len(x), p=probs)
    # ----------------------------------------------
    return idx
```

In [16]:
```
grader.check("q1d1")
```

Out[16]:

q1.d1 passed! 🍀

Q1.d2: Boltzmann algorithm implementation (5 points)

Evaluate the Boltzmann algorithm on the same MAB problem as above, for three values of the parameters τ: 0.01, 0.1, and 1. Use the driver code provided to plot their performances across N=20 runs as a function of the number of pulls.

Note: You can use action-value estimates for the Boltzmann distribution.

In [17]:
```
def boltzmann(
    bandit: Bandit,
    tau: float = 0.1,
    init_q: float = .0
    ) -> Tuple[list, list, list]:
  """
  .inputs:
    bandit: A bandit problem, instantiated from the above class.
    c: The additional term coefficient.
    init_q: Initial estimation of each arm's value.
  .outputs:
    rew_record: The record of rewards at each timestep.
    avg_ret_record: The average summation of rewards up to step t, where t goes
    we define `ret_T` = \sum^T_{t=0}{r_t}, `avg_ret_record` = ret_T / (1+T).
    tot_reg_record: The  regret up to step t, where t goes from 0 to n_pulls.
    opt_action_perc_record: Percentage of optimal arm selected.
  """
  # init q values (the estimates)
  q = np.array([init_q]*bandit.n_arm, dtype=float)

  ret = .0
  rew_record = []
  avg_ret_record = []
  tot_reg_record = []
  opt_action_perc_record = []

  true_action_rewards = -np.abs(bandit.theta-np.array(bandit.actual_toxicity_pr
  optimal_reward = np.max(true_action_rewards)
  optimal_action = np.argmax(true_action_rewards)

  for t in range(bandit.n_pulls):
    # ----------------------------------------------
    chosen_arm = boltzmann_policy(q, tau)
    reward = bandit.pull(chosen_arm)
    rew_record.append(reward)
```

```
    # update rule
    #   num_dose_selected will be automatically updated when the arm is pulled.

    q_a = q[chosen_arm]
    n_a = bandit.num_dose_selected[chosen_arm]
    q_a = q_a + (1./n_a) * (reward - q_a)
    q[chosen_arm] = q_a

    opt_action_perc_record.append(100*bandit.num_dose_selected[optimal_action]/

  returns = np.cumsum(rew_record)
  denoms = np.arange(len(returns))
  denoms += 1
  avg_ret_record = returns/denoms

  cumulative_optimal_rewards = denoms * optimal_reward
  tot_reg_record = cumulative_optimal_rewards - returns

  tot_reg_record.tolist()
  avg_ret_record.tolist()
  tot_reg_record.tolist()
    # ---------------------------------------------

  return rew_record, avg_ret_record, tot_reg_record, opt_action_perc_record
```

In [18]:
```
grader.check("q1d2")
```

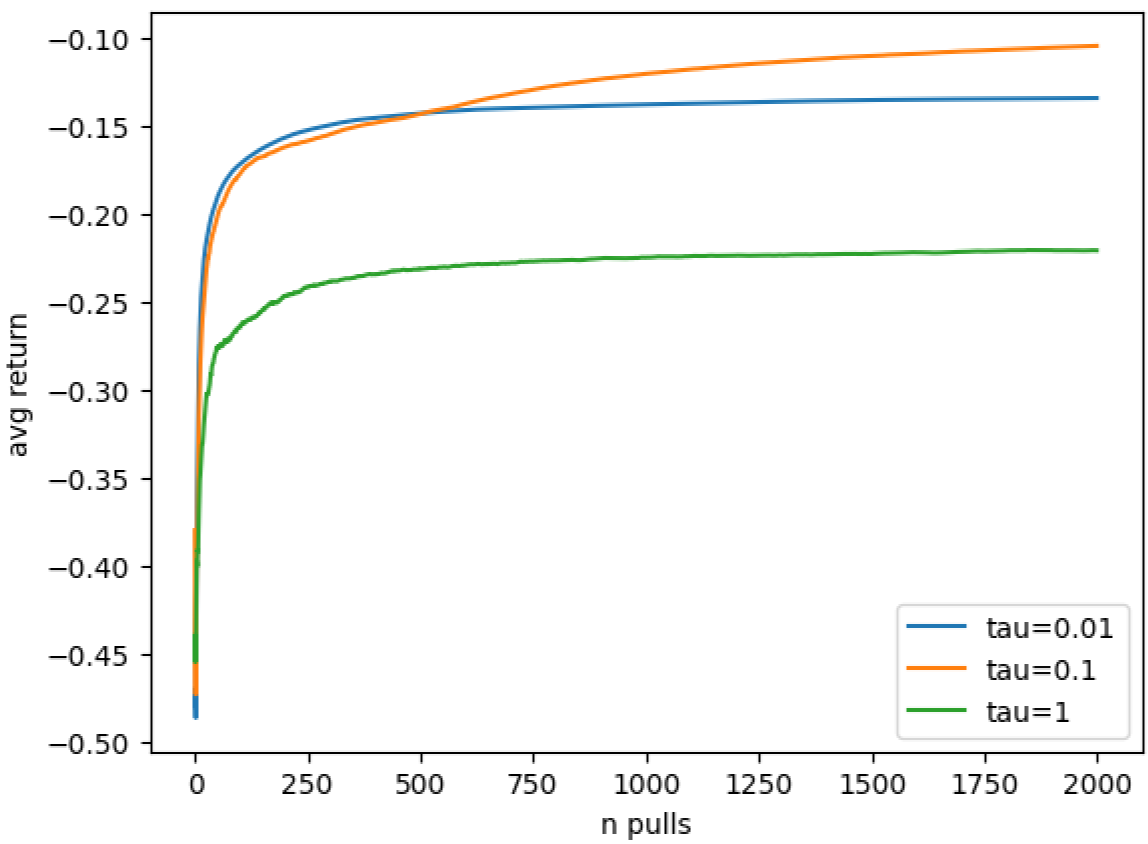
Out[18]:

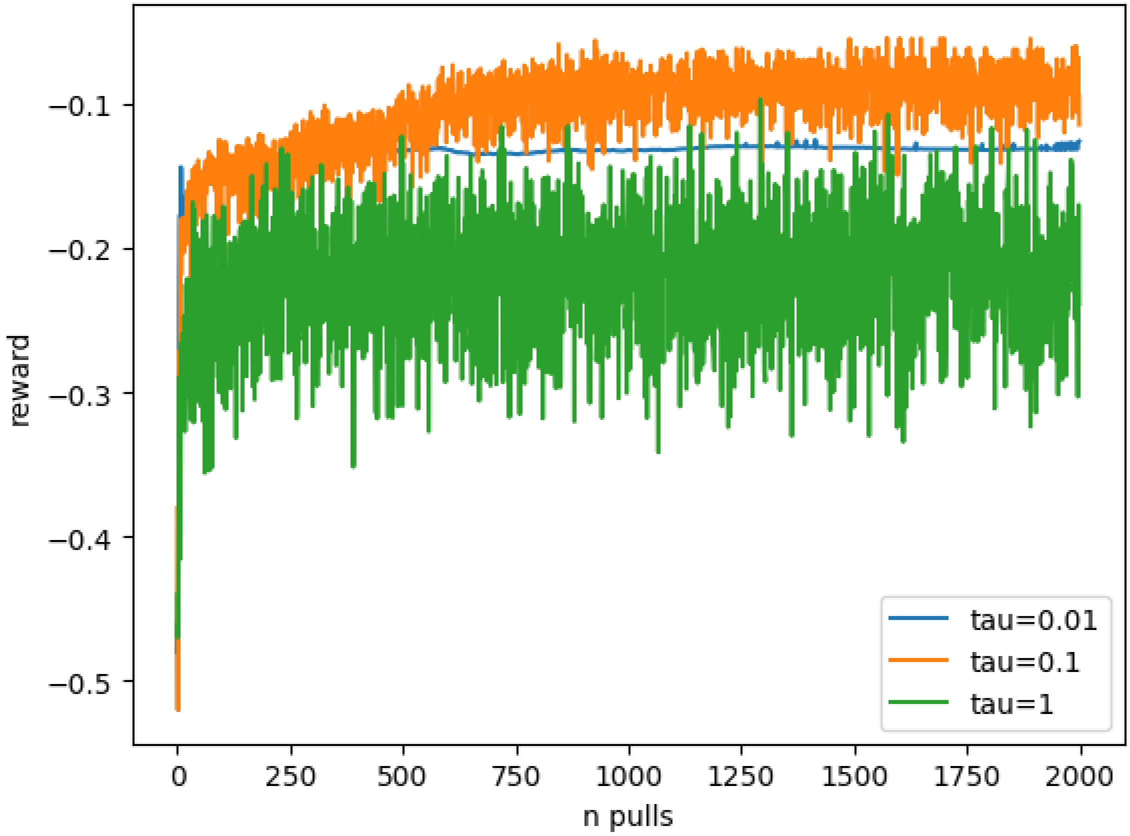
q1.d2 passed! ✨

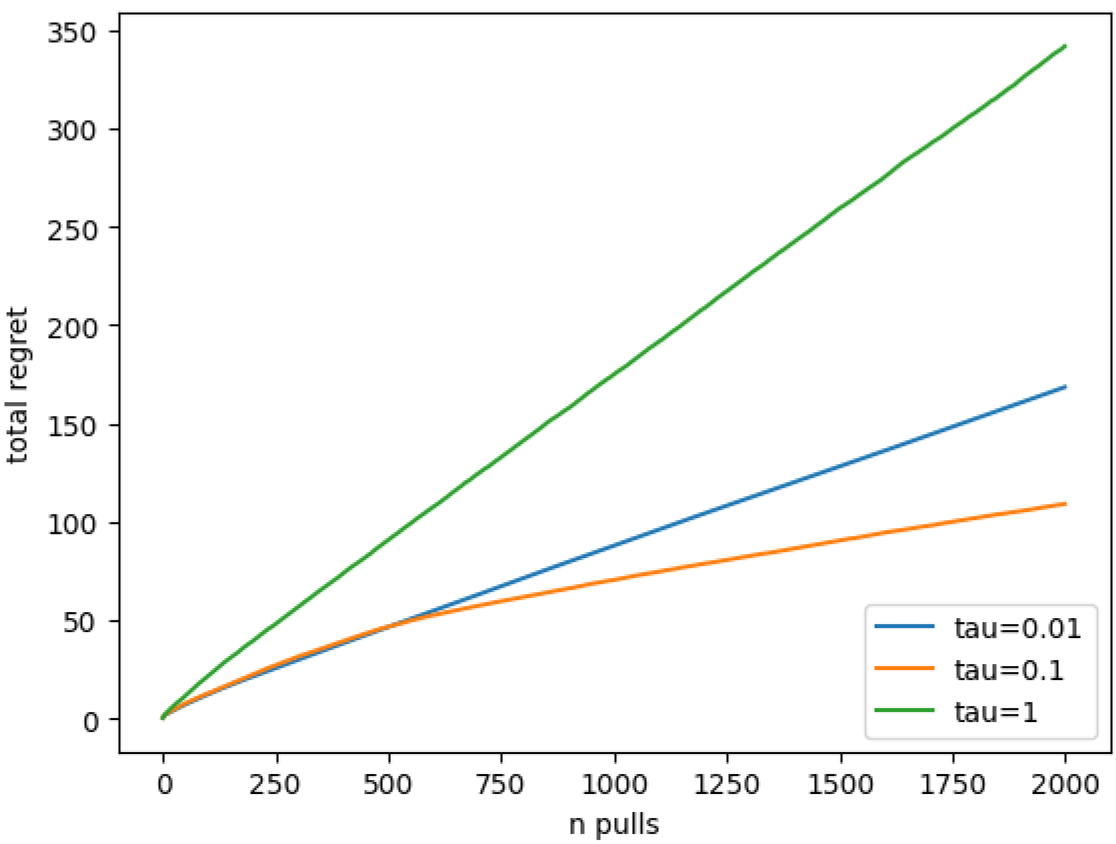
Q1.d3: Plotting the results (5 points)

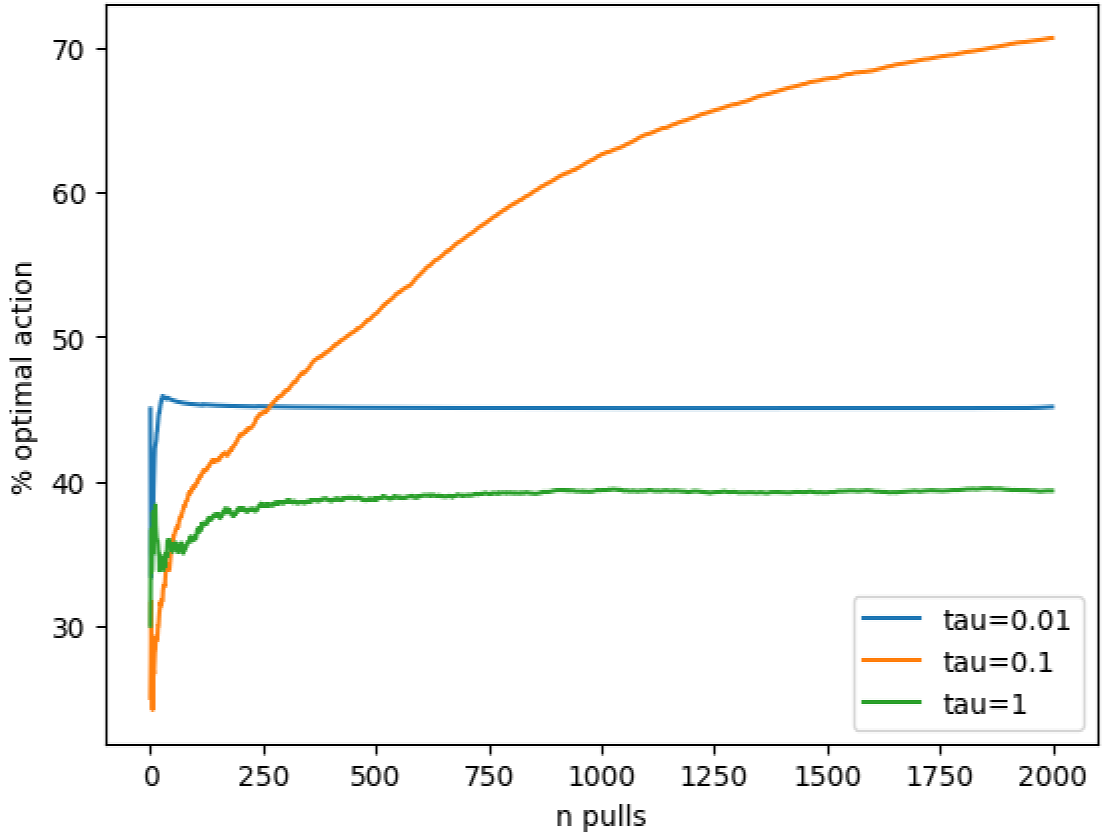
In [19]:
```
plt.figure(11)
plt.xlabel("n pulls")
plt.ylabel("avg return")
plt.figure(12)
plt.xlabel("n pulls")
plt.ylabel("reward")
plt.figure(13)
plt.xlabel("n pulls")
plt.ylabel("total regret")
plt.figure(14)
plt.xlabel("n pulls")
plt.ylabel("% optimal action")

N = 20
tot_reg_rec_best = 1e8
for tau in [0.01, 0.1, 1]:
  rew_rec = np.zeros(bandit.n_pulls)
  avg_ret_rec = np.zeros(bandit.n_pulls)
  tot_reg_rec = np.zeros(bandit.n_pulls)
  opt_act_rec = np.zeros(bandit.n_pulls)

  for n in range(N):
    bandit.init_bandit()
    rew_rec_n, avg_ret_rec_n, tot_reg_rec_n, opt_act_rec_n = boltzmann(bandit,
    rew_rec += np.array(rew_rec_n)
```

```
      avg_ret_rec += np.array(avg_ret_rec_n)
      tot_reg_rec += np.array(tot_reg_rec_n)
      opt_act_rec += np.array(opt_act_rec_n)

    # take the mean
    rew_rec /= N
    avg_ret_rec /= N
    tot_reg_rec /= N
    opt_act_rec /= N

    plt.figure(11)
    plt.plot(avg_ret_rec, label="tau={}".format(tau))
    plt.legend(loc="lower right")

    plt.figure(12)
    plt.plot(rew_rec, label="tau={}".format(tau))
    plt.legend(loc="lower right")

    plt.figure(13)
    plt.plot(tot_reg_rec, label="tau={}".format(tau))
    plt.legend(loc="lower right")

    plt.figure(14)
    plt.plot(opt_act_rec, label="tau={}".format(tau))
    plt.legend(loc="lower right")

    if tot_reg_rec[-1] < tot_reg_rec_best:
          boltzmann_dict = {
          'opt_act':opt_act_rec,
          'regret_list':tot_reg_rec,}
          tot_reg_rec_best = tot_reg_rec[-1]
```




reward
n pulls
tau=0.01
tau=0.1
tau=1
−0.1
−0.2
−0.3
−0.4
−0.5
0
250
500
750
1000
1250
1500
1750
2000


total regret
n pulls
tau=0.01
tau=0.1
tau=1
350
300
250
200
150
100
50
0
0
250
500
750
1000
1250
1500
1750
2000



Q1.d4: Analysis (5 points)

Explain the role of τ paramtere on the results.

- The parameter τ directly affects exploration vs exploitation because it controls the sharpness of the softmax, when tau is the smallest (0.01) in this case, then the softmax is very sharp and probability is concentrated on the greedy with the highest Q value we found (so far) which will lead to the most exploitatation (taking greedy actions), as we increase the value of τ we're incoraging more exploration by giving the exloratory actions more chance of being sampled, as we reach $\tau = 1$ we're doing a high exploration which will lead to a very slow convergence of the algorithm.

Q1.f: Gradient Bandits Algorithm (15 points)

Q1.f1: GB implementation (5 points)

Follow the lecture notes to implement the Gradient Bandits algorithm with and without the baseline.

In [20]:
```
def softmax(x):
    return np.exp(x) / np.sum(np.exp(x), axis=0)
```

In [21]:
```
def gradient_bandit(
    bandit: Bandit,
    alpha: float,
```

```
    use_baseline: bool = True,
    ) -> Tuple[list, list, list]:
  """
  .inputs:
    bandit: A bandit problem, instantiated from the above class.
    alpha: The learning rate.
    use_baseline: Whether or not use avg return as baseline.
  .outputs:
    rew_record: The record of rewards at each timestep.
    avg_ret_record: The average summation of rewards up to step t, where t goes
    we define `ret_T` = \sum^T_{t=0}{r_t}, `avg_ret_record` = ret_T / (1+T).
    tot_reg_record: The  regret up to step t, where t goes from 0 to n_pulls.
    opt_action_perc_record: Percentage of optimal arm selected.
  """
  # init h (the logits)
  h = np.array([0]*bandit.n_arm, dtype=float)

  ret = .0
  r_bar_t = 0
  rew_record = []
  avg_ret_record = []
  tot_reg_record = []
  opt_action_perc_record = []

  actions = np.eye(bandit.n_arm)
  true_action_rewards = -np.abs(bandit.theta-np.array(bandit.actual_toxicity_pr
  optimal_reward = np.max(true_action_rewards)
  optimal_action = np.argmax(true_action_rewards)
  for t in range(bandit.n_pulls):
    # ----------------------------------------------
    policy = softmax(h)
    chosen_arm = np.random.choice(len(h), p=policy)
    reward = bandit.pull(chosen_arm)
    rew_record.append(reward)

    r_bar_t = r_bar_t + (reward - r_bar_t) * 1/(t+1)
    probs_diff = actions[chosen_arm].flatten() - policy
    if use_baseline:
      h = h + alpha * (reward - r_bar_t) * probs_diff
    else:
      h = h + alpha * reward * probs_diff

    opt_action_perc_record.append(100*bandit.num_dose_selected[optimal_action]/

  returns = np.cumsum(rew_record)
  denoms = np.arange(len(returns))
  denoms += 1
  avg_ret_record = returns/denoms

  cumulative_optimal_rewards = denoms * optimal_reward
  tot_reg_record = cumulative_optimal_rewards - returns

  tot_reg_record.tolist()
  avg_ret_record.tolist()
  tot_reg_record.tolist()

    # ----------------------------------------------

  return rew_record, avg_ret_record, tot_reg_record, opt_action_perc_record
```

In [22]:
```
grader.check("q1f1")
```

Out[22]:

q1.f1 passed! 🎉

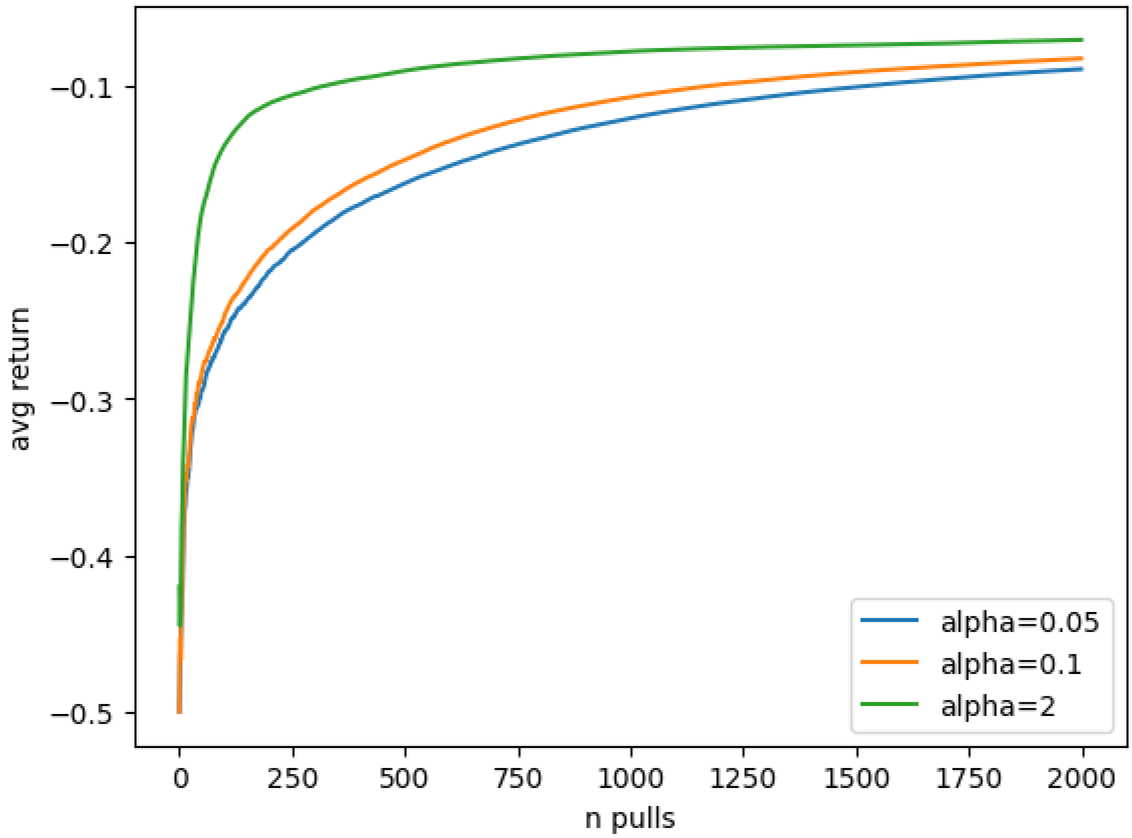
Q1.f2: Plotting the results (5 points)

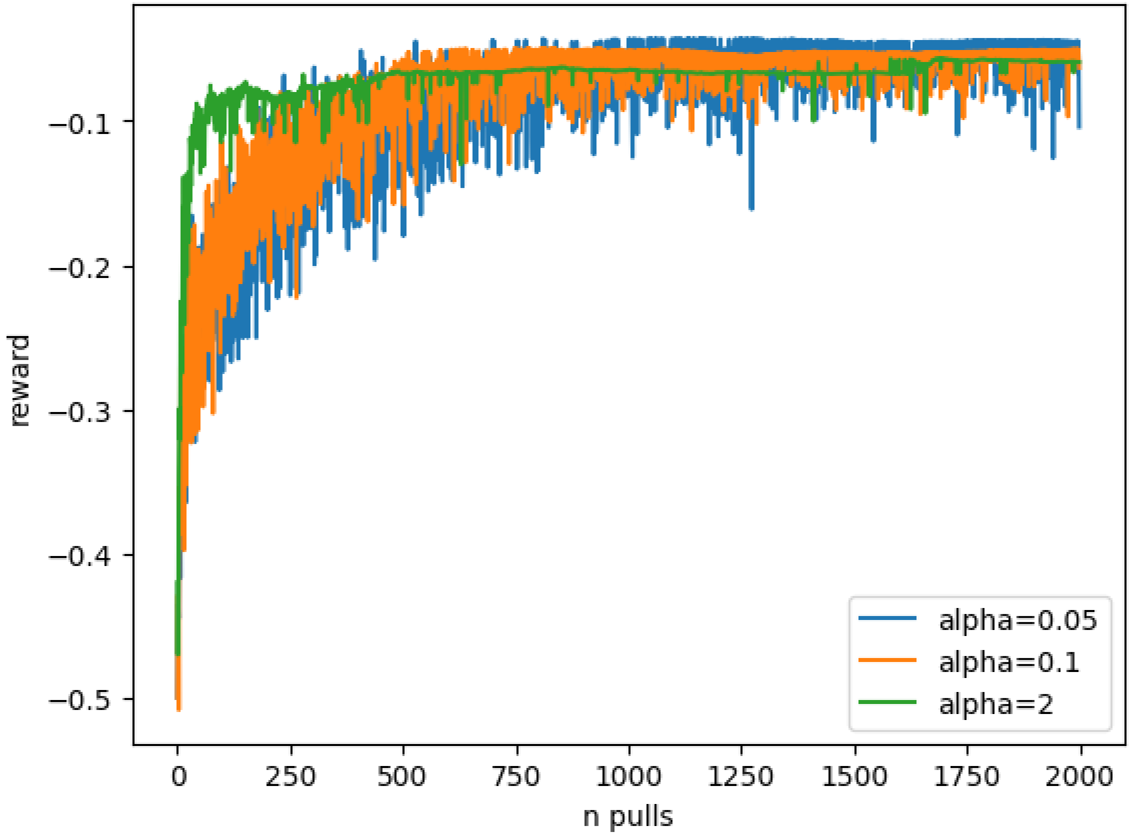
Evaluate the GB algorithm on the same MAB problem as above, for three values of the parameters α: 0.05, 0.1, and 2. Use the driver code provided to plot their performances.

With baseline:

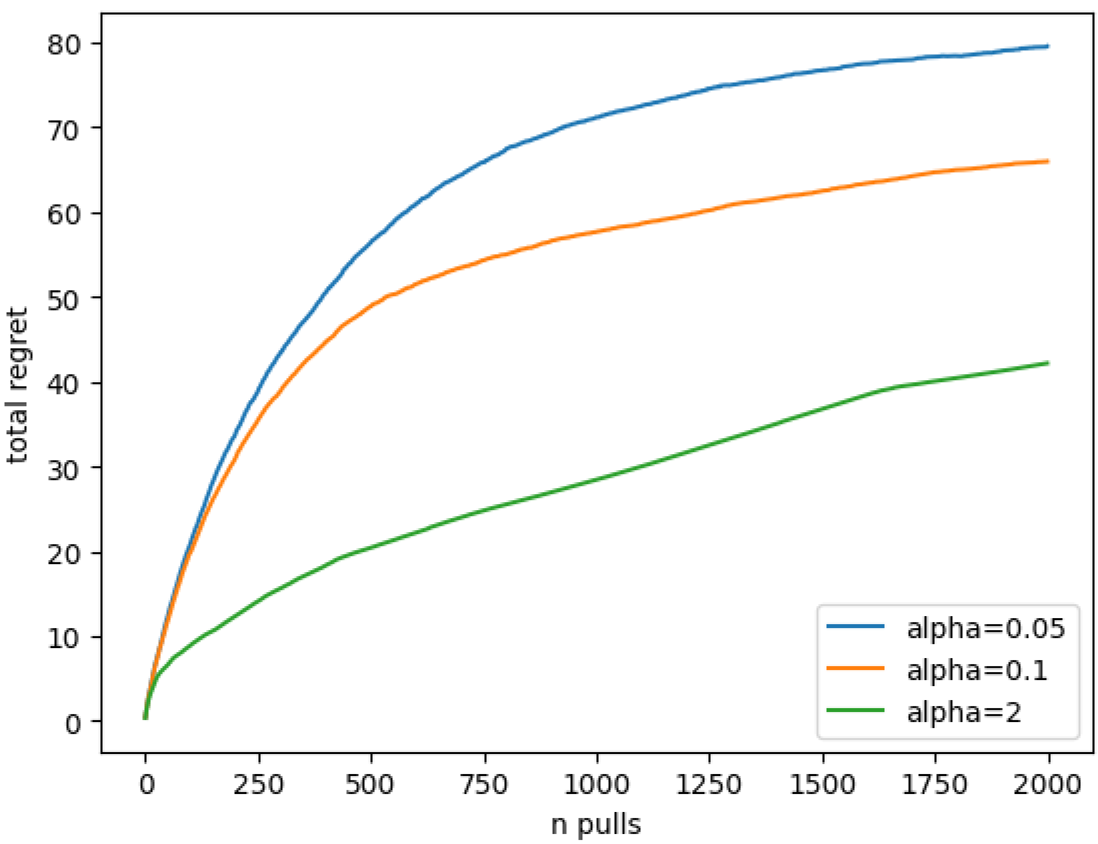
In [23]:
```
plt.figure(15)
plt.xlabel("n pulls")
plt.ylabel("avg return")
plt.figure(16)
plt.xlabel("n pulls")
plt.ylabel("reward")
plt.figure(17)
plt.xlabel("n pulls")
plt.ylabel("total regret")
plt.figure(18)
plt.xlabel("n pulls")
plt.ylabel("% optimal action")

N = 20
tot_reg_rec_best = 1e8
for alpha in [0.05, 0.1, 2]:
  rew_rec = np.zeros(bandit.n_pulls)
  avg_ret_rec = np.zeros(bandit.n_pulls)
  tot_reg_rec = np.zeros(bandit.n_pulls)
  opt_act_rec = np.zeros(bandit.n_pulls)

  for n in range(N):
    bandit.init_bandit()
    rew_rec_n, avg_ret_rec_n, tot_reg_rec_n, opt_act_rec_n = gradient_bandit(ba
    rew_rec += np.array(rew_rec_n)
    avg_ret_rec += np.array(avg_ret_rec_n)
    tot_reg_rec += np.array(tot_reg_rec_n)
    opt_act_rec += np.array(opt_act_rec_n)


  # take the mean
  rew_rec /= N
  avg_ret_rec /= N
  tot_reg_rec /= N
  opt_act_rec /= N


  plt.figure(15)
  plt.plot(avg_ret_rec, label="alpha={}".format(alpha))
  plt.legend(loc="lower right")

  plt.figure(16)
  plt.plot(rew_rec, label="alpha={}".format(alpha))
  plt.legend(loc="lower right")
```

```
plt.figure(17)
plt.plot(tot_reg_rec, label="alpha={}".format(alpha))
plt.legend(loc="lower right")

plt.figure(18)
plt.plot(opt_act_rec, label="alpha={}".format(alpha))
plt.legend(loc="lower right")

if tot_reg_rec[-1] < tot_reg_rec_best:
      gradient_bandit_dict = {
      'opt_act':opt_act_rec,
      'regret_list':tot_reg_rec,}
      tot_reg_rec_best = tot_reg_rec[-1]
```

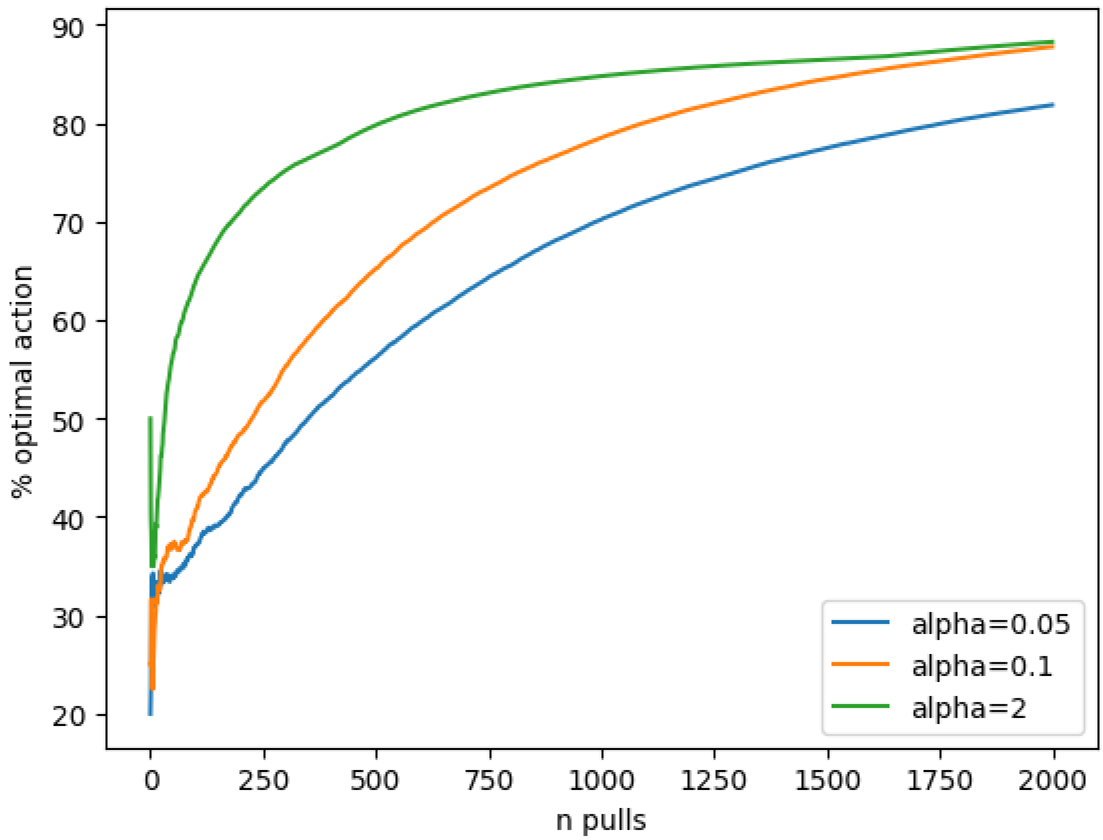




reward
n pulls
alpha=0.05
alpha=0.1
alpha=2



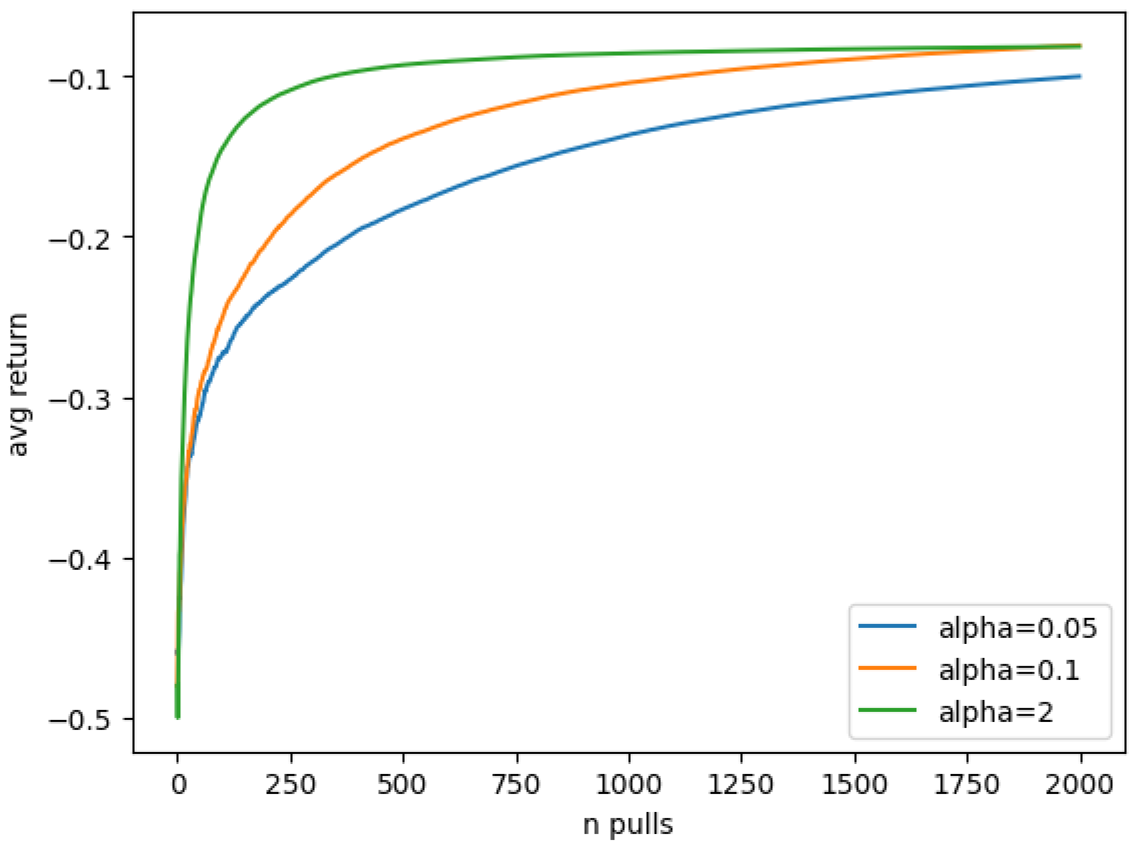
total regret
n pulls
alpha=0.05
alpha=0.1
alpha=2

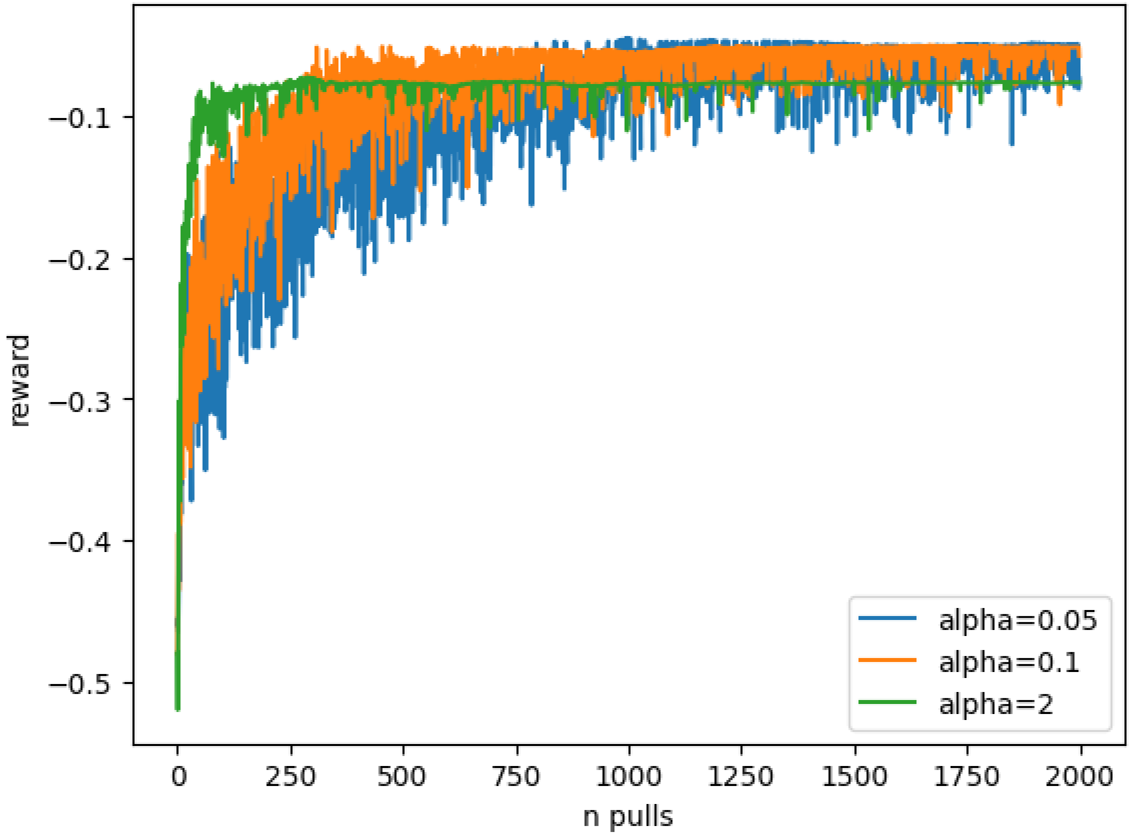


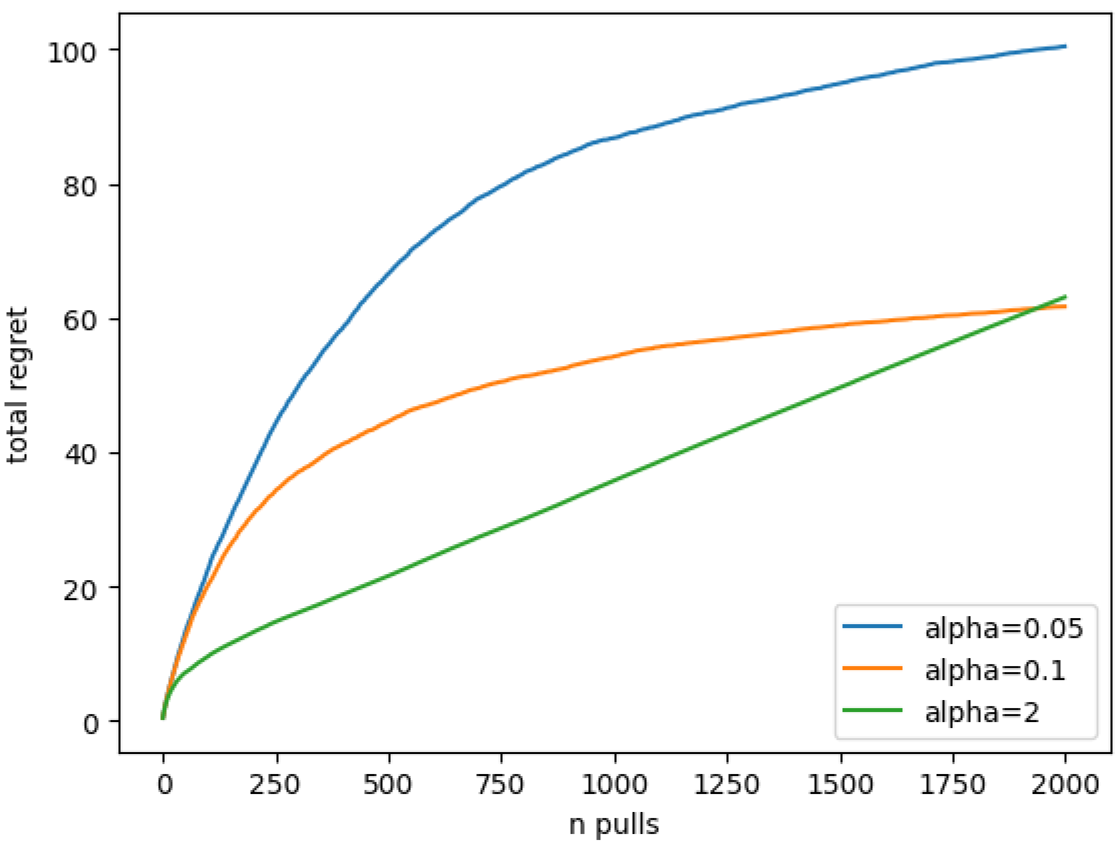
Without baseline:

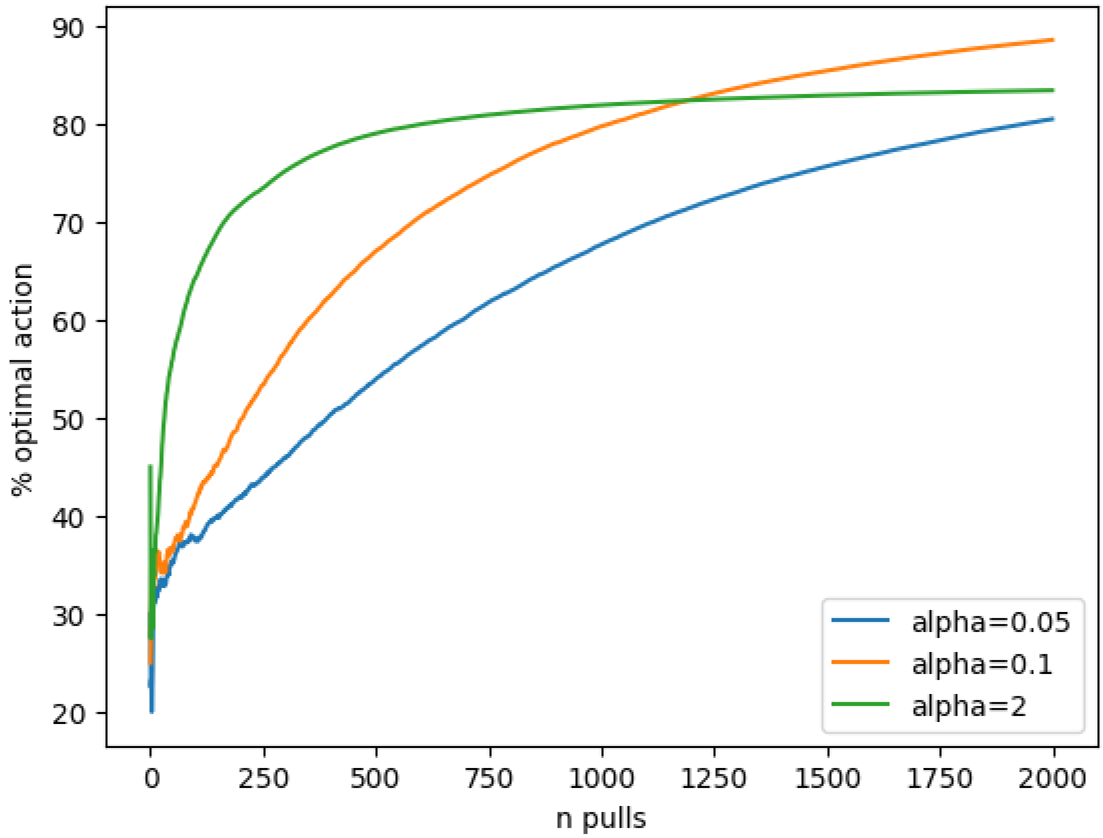
In [28]:
```
plt.figure(19)
plt.xlabel("n pulls")
plt.ylabel("avg return")
plt.figure(20)
plt.xlabel("n pulls")
plt.ylabel("reward")
plt.figure(21)
plt.xlabel("n pulls")
plt.ylabel("total regret")
plt.figure(22)
plt.xlabel("n pulls")
plt.ylabel("% optimal action")

N = 20
tot_reg_rec_best = 1e8
for alpha in [0.05, 0.1, 2]:
  rew_rec = np.zeros(bandit.n_pulls)
  avg_ret_rec = np.zeros(bandit.n_pulls)
  tot_reg_rec = np.zeros(bandit.n_pulls)
  opt_act_rec = np.zeros(bandit.n_pulls)

  for n in range(N):
    bandit.init_bandit()
    rew_rec_n, avg_ret_rec_n, tot_reg_rec_n, opt_act_rec_n = gradient_bandit(ba
    rew_rec += np.array(rew_rec_n)
    avg_ret_rec += np.array(avg_ret_rec_n)
    tot_reg_rec += np.array(tot_reg_rec_n)
    opt_act_rec += np.array(opt_act_rec_n)
```

```
    # take the mean
    rew_rec /= N
    avg_ret_rec /= N
    tot_reg_rec /= N
    opt_act_rec /= N

    plt.figure(19)
    plt.plot(avg_ret_rec, label="alpha={}".format(alpha))
    plt.legend(loc="lower right")

    plt.figure(20)
    plt.plot(rew_rec, label="alpha={}".format(alpha))
    plt.legend(loc="lower right")

    plt.figure(21)
    plt.plot(tot_reg_rec, label="alpha={}".format(alpha))
    plt.legend(loc="lower right")

    plt.figure(22)
    plt.plot(opt_act_rec, label="alpha={}".format(alpha))
    plt.legend(loc="lower right")

    if tot_reg_rec[-1] < tot_reg_rec_best:
          gradient_bandit_dict = {
          'opt_act':opt_act_rec,
          'regret_list':tot_reg_rec,}
          tot_reg_rec_best = tot_reg_rec[-1]
```




reward
−0.1
−0.2
−0.3
−0.4
−0.5
0
250
500
750
1000
1250
1500
1750
2000
n pulls
alpha=0.05
alpha=0.1
alpha=2


total regret
100
80
60
40
20
0
0
250
500
750
1000
1250
1500
1750
2000
n pulls
alpha=0.05
alpha=0.1
alpha=2



Q1.f3: Analysis (5 points)

Explain the role of α and the baseline on the results.

Role of α

- Depending on using baseline or not the role of alpha changes, [But generally alpha controlls how fast the algorithm can converge (consider it as learning speed)].
- The difference between α = 0.1 and α = 0.2 is small (asymptotically) but large initially.
  - It seems having α = 0.1 yields the best result, 'medium' value of learning rate when using no baseline, and α = 2 gives best result when using baseline, having a very low α slows the convergence, in both cases α = 0.05 yileded worst results because it is slow.

Role of baseline

It seems from the above results that the baselines achives two things:

- It improves the results for the higher learning rate (α =2). Makes training with larger learning rate perform better (more stability).
- It reduces the variance in the rewards.

The degree to which alpha and baseline control exploration vs exploitation depends, because there are many factors in here and it depends on both values, also in both cases exploration is encoraged by the initilization.

Q1.g: Final comaprison (10 points)

Q1.g1: plots (5 points)

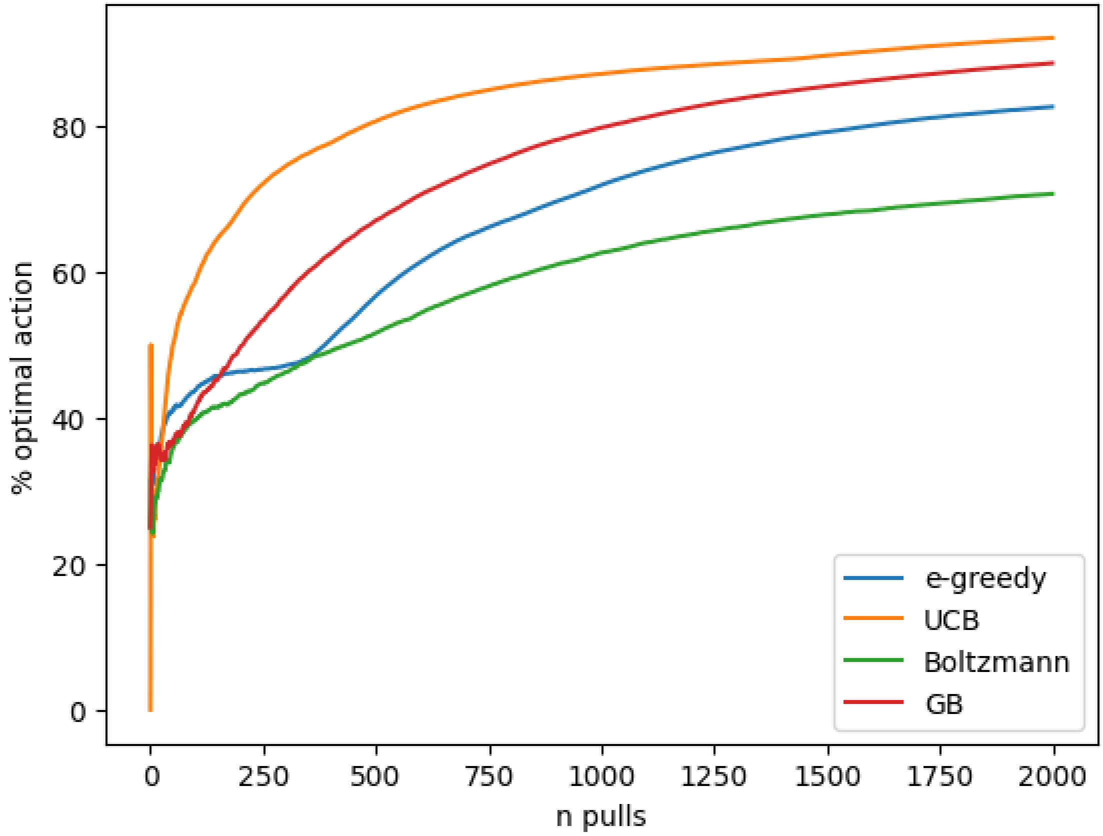
Compare the performance of ϵ-greedy, UCB, Boltzmann algorithm, and Gradient Bandit algorithm in a single plot as measured by the average reward and total regret.

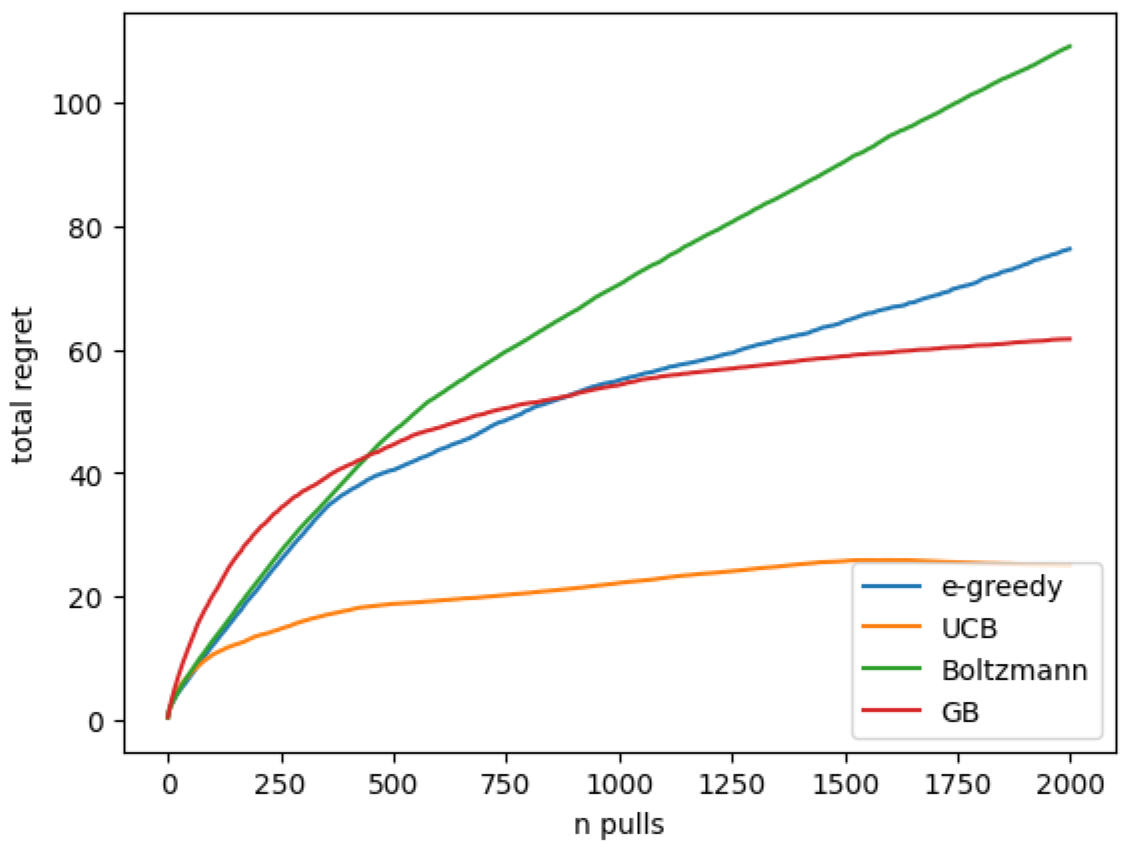
In [29]:

```
plt.figure(23)
plt.plot(ep_greedy_dict["opt_act"], label="e-greedy")
plt.legend(loc="lower right")
plt.plot(ucb_dict["opt_act"], label="UCB")
plt.legend(loc="lower right")
plt.plot(boltzmann_dict["opt_act"], label="Boltzmann")
plt.legend(loc="lower right")
plt.plot(gradient_bandit_dict["opt_act"], label="GB")
plt.legend(loc="lower right")
plt.xlabel("n pulls")
plt.ylabel("% optimal action")

plt.figure(24)
plt.plot(ep_greedy_dict["regret_list"], label="e-greedy")
plt.legend(loc="lower right")
plt.plot(ucb_dict["regret_list"], label="UCB")
plt.legend(loc="lower right")
plt.plot(boltzmann_dict["regret_list"], label="Boltzmann")
plt.legend(loc="lower right")
plt.plot(gradient_bandit_dict["regret_list"], label="GB")
plt.legend(loc="lower right")
plt.xlabel("n pulls")
plt.ylabel("total regret")
```

Out[29]:

```
Text(0, 0.5, 'total regret')
```





Q1.g2: Analysis (5 points)

Compare all the algorithms in terms of their performance.

From the plots the order from best to worst in terms of both highest % of optimal action and lowest regret is: UCB followed by Gradient Banidts, followed by ϵ-greedy and finally Boltzman exploration.

In [30]:
```
plt.close('all')
```