1. **什么是人工智能？它的研究目标是什么？**

人工智能，指由人制造出来的机器所表现出来的智能。通常人工智能是指通过普通计算机程序来呈现人类智能的技术。人工智能也指出研究这样的智能系统是否能够实现，以及如何实现。

人工智能，即智能科学技术研究追求的目标是构建智能体像人类一样行动，并且要合理的行动，像人一样思考，并且要合理思考。

2. **人工智能有哪几个主要的学派？各自的特点是什么？**

对人工智能研究影响较大的的主要有符号主义、连接主义和行为主义三大学派。

符号主义学派：

符号主义是一种基于逻辑推理的智能模拟方法，其原理主要为物理符号系统假设和有限合理性原理，长期以来，一直在人工智能中处于主导地位。

符号主义学派认为人工智能源于数学逻辑。该学派认为人类认知和思维的基本单元是符号，而认知过程就是在符号表示上的一种运算。符号主义致力于用计算机的符号操作来模拟人的认知过程其，实质就是模拟人的左脑抽象逻辑思维，通过研究人类认知系统的功能机理，用某种符号来描述人类的认知过程，并把这种符号输入到能处理符号的计算机中，从而模拟人类的认知过程，实现人工智能。

连接主义学派：

连接主义又称为仿生学派，是一种基于神经网络及网络间的连接机制与学习算法的智能模拟方法。其原理主要为神经网络和神经网络间的连接机制和学习算法。这一学派认为人工智能源于仿生学，特别是人脑模型的研究。

连结主义学派从神经生理学和认知科学的研究成果出发，把人的智能归结为人脑的高层活动的结果，强调智能活动是由大量简单的单元通过复杂的相互连接后并行运行的结果。其中人工神经网络就是其典型代表性技术。

行为主义学派：

行为主义又称进化主义(Evolutionism)或控制论学派(Cyberneticsism)，是一种基于“感知——行动”的行为智能模拟方法。

行为主义最早来源于 20 世纪初的一个心理学流派，认为行为是有机体用以适应环境变化的各种身体反应的组合，它的理论目标在于预见和控制行为。维纳和麦洛克等人提出的控制论和自组织系统以及钱学森等人提出的工程控制论和生物控制论，影响了许多领域。控制论把神经系统的工作原理与信息理论、控制理论、逻辑以及计算机联系起来。早期的研究工作重点是模拟人在控制过程中的智能行为和作用，对自寻优、自适应、自校正、自镇定、自组织和自学习等控制论系统的研究，并进行“控制动物”的研制。到 60、70 年代，上述这些控制论系统的研究取得一定进展，并在 80 年代诞生了智能控制和智能机器人系统。

3. **达特茅斯会议参加者对人工智能各自有什么学术贡献？**

约翰·麦卡锡：

在 1956 年的达特矛斯会议上提出了“人工智能”这个概念。

麦卡锡发明了 LISP 并于 1960 年将其设计发表在《ACM 通讯》上。他帮助推动了麻省理工学院的 MAC 项目。然而，他在 1962 年了离开麻省理工学院，前往斯坦福大

学并在那里协助建立了斯坦福人工智能实验室，成为 MAC 项目多年来的一个友好的竞争对手。

为了减少计算机需要考虑的棋步，麦卡锡发明了著名的α－β 搜索法，这一关键问题的解决有效减少了计算量，使其至今仍是解决人工智能问题中一种常用的高效方法。

马文·闵斯基：

在 1955 年的达特矛斯会议上，麦卡锡与马文·明斯基共同提出的“人工智能”这个概念。他还是个发明家，拥有捻船缝机和桔汁冷冻机两项专利。

克劳德·香农：

香农在《贝尔系统技术杂志》（Bell System Technical Journal）上连载发表了影像深远的论文《通讯的数学原理》。1949 年，香农又在该杂志上发表了另一著名论文《噪声下的通信》。在这两篇论文中，香农解决了过去许多悬而未决的问题：阐明了通信的基本问题，给出了通信系统的模型，提出了信息量的数学表达式，并解决了信道容量、信源统计特性、信源编码、信道编码等一系列基本技术问题。两篇论文成为了信息论的基础性理论著作。

4. 人工智能有哪些主要的研究和应用领域？你更喜欢哪个领域，综述其成就，剖析其问题，并提出您对该领域发展的见解？

人工智能的主要研究领域：

机器感知：机器视觉，机器听觉，自然语言理解，机器翻译

机器思维：机器推理

机器学习：符号学习，连接学习，强化学习，模式学习，迁移学习，深度学习

机器行为：智能控制

人工智能的应用领域：

智能机器：智能机器人，机器智能

机器博弈：自动定理证明

专家系统：智能决策，智能检索，智能 CAD

智能交通：智能电力，智能产品，智能建筑

我更喜欢智能交通领域，目前许多汽车厂家都已经实现了初级的自动驾驶和辅助驾驶功能。特斯拉的 autopilot 已经可以运用在大部分场合，但由于技术的限制，暂时还不能实现完全的无人驾驶功能。

我认为无人驾驶技术前景非常的广泛实用，而且近几年发展的很快，相信很快就能实现真正的无人驾驶。

2.6 解：

a. 有无穷多个代理程序可以实现给定的代理函数，例如，一个感知程序闪光的代理程序，可以把光线的明暗变化映射到一些输出，比如，整数。这个感知程序可以按照光线明暗变化的序列，将它映射为整数，也可以直接映射为代表其自身的整数。

b. 有。一个函数，其功能测量是判断一个程序是否停止不能作为程序实现。

c. 能。此时，改变映射的同时必须改变程序。

d. 有2^n种可能的代理程序，若每个状态有 a 种可能的选择，则一共有a^{2^n}种可能的代理程序。

e. 机器速度提高不会改变代理函数。

2.7 解：

a. 基于目标的 agent:

```
let
state
model
goals
action
    define (goal-based-agent percept)
    set! state (update-state state action percept model)
    let
    action-sequences; and, if so, how to do so without a utility
    function: evaluate them against the performance measure?
    action-sequence (search goals state)
        return (first action-sequence)
```

b. 基于效用的 agent:

```
let
state
model
goals
action
    define (utility-based-agent percept)
    set! state (update-state state action percept model)
    let
    probabilities (map probability goals)
    utilities (map utility goals)
    let
    expected-utilities (map * probabilities utilities)
    goal-of-maximum-expected-utility (max goals expected-utilities)
    action-sequence (search goal-of-maximum-expected-utility state)
        return (first action-sequence)
```

2.8 解：

（此部分参考了网上的代码实现）

```
(use debug
     foof-loop
     lolevel
     srfi-1
     srfi-8
     srfi-13
     srfi-69
     vector-lib)

(define (simulate environment)
  (loop ((while (environment)))))

(define (compose-environments . environments)
  (lambda ()
    (every identity (map (lambda (environment)
                           (environment))
                         environments))))

(define (make-performance-measuring-environment
         measure-performance
         score-update!)
  (lambda () (score-update! (measure-performance))))

(define (make-step-limited-environment steps)
  (let ((current-step 0))
    (lambda ()
      (set! current-step (+ current-step 1))
      (< current-step steps))))

;;; What about pairs of objects and optional display things.
(define make-debug-environment
  (case-lambda
   ((object) (make-debug-environment object pp))
   ((object display)
    (lambda () (display object)))))

(define (vacuum-world-display world)
  (pp
   (vector-append '#(world)
                  (vector-map
                   (lambda (i clean?)
                     (if clean? 'clean 'dirty))
                   world))))
```

```
(define clean #t)
(define clean? identity)

(define dirty #f)
(define dirty? (complement clean?))

(define left 0)
(define left? zero?)

(define right 1)
(define right? (complement zero?))

(define make-vacuum-world vector)

(define vacuum-world-location vector-ref)

(define vacuum-world-location-set! vector-set!)

(define-record vacuum-agent
 location
 score
 program)

(define-record-printer vacuum-agent
 (lambda (vacuum-agent output)
   (format output
          "#(agent ~a ~a)"
          (if (left? (vacuum-agent-location vacuum-agent))
              'left
              'right)
          (vacuum-agent-score vacuum-agent))))

(define (make-vacuum-environment world agent)
 (lambda ()
   (let* ((location (vacuum-agent-location agent))
          (action ((vacuum-agent-program agent)
                   location
                   (vacuum-world-location world location))))
     (case action
       ((left) (vacuum-agent-location-set! agent left))
       ((right) (vacuum-agent-location-set! agent right))
       ((suck) (vacuum-world-location-set! world location clean))
       (else (error (string-join
                     "make-vacuum-environment --"
```

```
                  "Unknown action")
                 action))))))

(define (reflex-vacuum-agent-program location clean?)
  (if clean?
      (if (left? location)
          'right
          'left)
      'suck))

(define make-reflex-vacuum-agent
  (case-lambda
   ((location)
    (make-reflex-vacuum-agent location reflex-vacuum-agent-program))
   ((location program)
    (make-vacuum-agent
     location
     0
     program))))

(define (make-vacuum-performance-measure world)
  (lambda ()
    (vector-count (lambda (i square) (clean? square)) world)))

(define (make-vacuum-score-update! agent)
  (lambda (score)
    (vacuum-agent-score-set! agent (+ (vacuum-agent-score agent)
                                      score))))

(define simulate-vacuum
  (case-lambda
   ((world agent) (simulate-vacuum world agent 1000))
   ((world agent steps)
    (simulate
     (compose-environments
      (make-step-limited-environment steps)
      (make-performance-measuring-environment
       (make-vacuum-performance-measure world)
       (make-vacuum-score-update! agent))
      (make-debug-environment agent)
      (make-debug-environment world vacuum-world-display)
      (make-vacuum-environment world agent)))
    (vacuum-agent-score agent))))
```

```
(simulate-vacuum (make-vacuum-world dirty clean)
                 (make-reflex-vacuum-agent
                  left
                  (lambda (location clean?)
                    'right))
                 10)
```

2.9 解：

（此部分参考了网上的代码实现）

```
(use aima-vacuum
    test)

(let ((worlds
      (list (make-world clean clean)
            (make-world clean clean)
            (make-world clean dirty)
            (make-world clean dirty)
            (make-world dirty clean)
            (make-world dirty clean)
            (make-world dirty dirty)
            (make-world dirty dirty)))
     (agents
      (list (make-reflex-agent left)
            (make-reflex-agent right)
            (make-reflex-agent left)
            (make-reflex-agent right)
            (make-reflex-agent left)
            (make-reflex-agent right)
            (make-reflex-agent left)
            (make-reflex-agent right))))
  (let* ((scores (map simulate-vacuum worlds agents))
         (average-score (/ (apply + scores) 8)))
    (test
     "Scores for each configuration"
     scores
     '(2000 2000 1998 1999 1999 1998 1996 1996))
    (test
     "Average overall score"
     1998.25
     average-score)))
```

3.1 解：

（汉诺塔问题）：采用递归实现：

```
def hano(n,x,y,z):  #将 x 上的盘子借助 y 移动到 z 上
    if n == 1:
        print(x,"-->",z)
    else:
        hano(n-1,x,z,y)      #将前 n-1 个盘子从 x 移动到 y 上
        print(x,"-->",z)     #将最底下的最后一个盘子从 x 移动到 z 上
        hano(n-1,y,x,z)      #将 y 上的 n-1 个盘子借助 x 移动到 z 上
n = int(input("请输入汉诺塔层数："))

hano(n,'X','Y','Z')
```

3.2 解：

（过河问题）：采用深度优先搜索：

```
name = ["farmer", "wolf", "sheep", "grass"]
scheme_count = 0

# 完成局面
def is_done(status):
    return status[0] and status[1] and status[2] and status[3]

# 生成下一个局面的所有情况
def create_all_next_status(status):
    next_status_list = []

    for i in range(0, 4):
        if status[0] != status[i]: # 和农夫不同一侧?
            continue

        next_status = [status[0],status[1],status[2],status[3]]
        # 农夫和其中一个过河，i 为 0 时候，农夫自己过河。
        next_status[0] = not next_status[0]
        next_status[i] = next_status[0] # 和农夫一起过河

        if is_valid_status(next_status):
            next_status_list.append(next_status)

    return next_status_list

# 判断是否合法的局面
def is_valid_status(status):
```

```
    if status[1] == status[2]:
        if status[0] != status[1]:
            # 狼和羊同侧，没有人在场
            return False

    if status[2] == status[3]:
        if status[0] != status[2]:
            # 羊和草同侧，没有人在场
            return False

    return True

def search(history_status):
    global scheme_count
    current_status = history_status[len(history_status) - 1]

    next_status_list = create_all_next_status(current_status)
    for next_status in next_status_list:
        if next_status in history_status:
            # 出现重复的情况了
            continue


        #中序遍历所有可到达的状态
        history_status.append(next_status)

        if is_done(next_status):
            scheme_count += 1
            print("scheme " + str(scheme_count) + ":")
            print_history_status(history_status)
        else:
            search(history_status)

        history_status.pop()



def readable_status(status, is_across):
    result = ""
    for i in range(0,4):
        if status[i] == is_across:
            if len(result) != 0:
                result += ","
            result += name[i]
```

```
    return "[" + result + "]"

#打印结果
def print_history_status(history_status):
    for status in history_status:
        print(readable_status(status,     False)     +     "---->----->"     +
readable_status(status, True))

if __name__ == "__main__":
    # 初始局面
    status = [False, False, False, False]
    # 局面队列
    history_status = [status]

    search(history_status)

    print("finish search, find " + str(scheme_count) + " scheme")
```