Student who wrote the solution to this question: Zimo Li, Yu-Hsuan Chuang, Jiahong Zhai
Students who read this solution to verify its clarity and correctness: Zimo Li, Yu-Hsuan Chuang, Jiahong Zhai

1. (a) Since we are trying to maximize the average grade of the projects, and the number of projects n is provided, this problem is equivalent to maximizing the total grade of the projects.

   Let $K(i, h)$ = value of the maximum total grade for projects $1, 2, \ldots, i$ and total hours h spend on n projects. If $i = 0$, then there are no projects and the grade can only be 0. If $h = 0$, then we have no hours spend on any of the projects, and we assume that project i can still get grade $f_i(0)$ if the project is not done at all.

   Consider the subproblem $K(i, h)$, there are $h+1$ possible cases for the last project i, we can spend k hours on this project where k ranges from 0 to h. And $K(i-1, h-k)$ would be a maximum total grade for projects $1, 2, \ldots, i-1$ and total hours $h-k$. Therefore

$$K(i,h) = \begin{cases} 0, & \text{if } i = 0 \\ \displaystyle\sum_{j=1}^{i} f_j(0), & \text{if } h = 0 \\ \displaystyle\max_{0 \le k \le h} \big[f_i(k) + K(i-1, h-k)\big], & \text{if } i > 0 \text{ and } h > 0 \end{cases}$$

   Pseudocode:

```
01 Knapsack(f1, ..., fn, H):
02   for h = 0 to H:
03     K(0,h) <- 0
04   for i = 0 to n:
05     K(i,0) <- f1(0) + ... + fi(0)
06   for i = 1 to n:
07     for h = 1 to H:
08       max <- 0
09       for k = 0 to h:
10         if fi(k) + K(i - 1, h - k) > max:
11           max <- fi(k) + K(i - 1, h - k)
12       K(i, h) <- max
13   return K(n, H) / n
```

   This algorithm has a running time of $O(nH^2)$. Since it takes $O(H)$ to find the maximum grade of for $K(i, h)$ at line 9, and there are $n \cdot H$ such entries in total (i.e. the for-loops at line 6 and line 7).

(b) Pseudocode:

```
Knapsack(f1, ..., fn, H):
  for h = 0 to H:
    K(0,h) <- 0
  for i = 0 to n:
    K(i,0) <- f1(0) + ... + fi(0)
  for i = 1 to n:
    for h = 1 to H:
      max <- 0
      for k = 0 to h:
        if fi(k) + K(i - 1, h - k) > max:
          max <- fi(k) + K(i - 1, h - k)
      K(i, h) <- max
  S <- emptyset(), i <- n, h <- H
  while i > 0 and h > 0:
    for k = 0 to h:
      if K(i, h) = fi(k) + K(i - 1, h - k):
        S.insert(i)
        i <- i - 1
        h <- h - k
        break
  return S
```

Student who wrote the solution to this question: Zimo Li, Yu-Hsuan Chuang, Jiahong Zhai
Students who read this solution to verify its clarity and correctness: Zimo Li, Yu-Hsuan Chuang, Jiahong Zhai

2. (a) Let $M(i,j)$ = the minimum cost of joining cars $i, i+1, \ldots, j$. Then $M(1,n)$ gives the minimum cost of joining all the cars. We define $cost(i,j)$ as the square root of the total weight of cars i to j. Consider the very last join before all cars are joined, there are two segments left, then there are $j-i$ different possibilities for how the cars are partitioned between the two segments, and we can solve the minimum cost of joining cars in these segments recursively.

   If $i=j$, we have only one car and it is already joined, so the cost is 0. If $i=j-1$, we have two cars, the minimum cost is the minimum of the square roots of their weight. Then

$$C(i,j) = \begin{cases} 0, & \text{if } i=j \\ \min\limits_{i<=k<j} \left[\min\left(tsqrt(i,k),\ tsqrt(k+1,j)\right) + C(i,k) + C(k+1,j)\right], & \text{if } i! = j \end{cases}$$

   Pseudocode:

```
Connect(w[n]):
  for i = 1 to n:
    M(i, i) <- 0
  for i = 1 to n - 1:
    M(i, i + 1) <- min(sqrt(w[i]), sqrt(w[i + 1]))
  for i = 1 to n:
    for j = i to n:
      minimum <- 0
      for k = i to j:
        current = min(cost(i, k), cost(k + 1, j)) + M(i, k) + M(k + 1, j)
        if minimum > current:
          minimum <- current
      M(i, j) <- minimum
  return M(1, n)
```

   The algorithm has a running time of $O(n^3)$ as there are $(1/2)n^2$ subproblems in the memoization matrix and each entry takes $O(n)$ time to solve.

(b) We can trace back in the $M(i, j)$ matrix, if we find a partition with minimum cost at car k, we append the join pair $(k, k+1)$ to a list. In the end, reverse the list and we will get the optimal order of joining.

Pseudocode:

```
Connect(w[n]):
  for i = 1 to n:
    M(i, i) <- 0
  for i = 1 to n - 1:
    M(i, i + 1) <- min(sqrt(w[i]), sqrt(w[i + 1]))
  for i = 1 to n:
    for j = i to n:
      minimum <- 0
      for k = i to j:
        current = min(cost(i, k), cost(k + 1, j)) + M(i, k) + M(k + 1, j)
        if minimum > current:
          minimum <- current
      M(i, j) <- minimum
  return Order(i, j, M).reverse()

Order(i, j, M)
  joins <- []
  while i != j:
   for k = i to j:
     if M(i, j) = min(cost(i, k), cost(k + 1, j)) + M(i, k) + M(k + 1, j):
       joins.append((i, j))
       joins.append(Order(i, k))
       joins.append(Order(k + 1, j))
  return joins
```

Student who wrote the solution to this question: Zimo Li, Yu-Hsuan Chuang, Jiahong Zhai
Students who read this solution to verify its clarity and correctness: Zimo Li, Yu-Hsuan Chuang, Jiahong Zhai

3. (a) The greedy algorithm is not correct when the chosen longest nonempty prefix w should actually split into 2 parts and one belongs to x and the other one belongs to y, despite the prefix w can also be matched with only one of x or y. For example, when z = '$AABBBCACC$', x = '$AABCA$', and y = '$BBCC$', the correct z will be 'AA' from x + 'BB' from y + 'BCA' from x + 'CC' from y. However, if we use greedy algorithm, we will first get 'AAB' from x + 'BBC' from y, and then the rest of z 'ACC' cannot be matched with x and y in the order. Therefore, choosing the longest nonempty prefix will cause the matching problem which makes the program return false instead of true.

(b) We define $S(i, j)$ as a subproblem that returns true iff the prefix of z of length $i + j$ is the interleaving of the prefix of x of length i and the prefix of y of length j. If the i-th character of x is the $(i + j)$-th character of z, then whether or not the interleaving is true depends on if the first $i - 1$ characters of x and the first j characters of y can produce the interleaving of the first $i + j - 1$ characters of z. In other words, $S(i, j)$ depends on $S(i - 1, j)$. Similarly, $S(i, j)$ also depends on $S(i, j - 1)$ if the j-th character of y is the $(i + j)$-th character of z. Then

$$S(i, j) = [S(i - 1, j) \text{ and } x_i = z_{i+j}] \text{ or } [S(i, j) \text{ and } y_j = z_{i+j}]$$

Pseudocode:

```
hash_xy = {}
def interleaving(x,y,z):
    if (len(x) + len(y) != len(z)):
        return False
    for i in range(len(x) + 1):
        for j in range(len(y) + 1):
            hash_xy[(i,j)] = False #initialize every path to False
            if ((i,j) == (0,0)):
                hash_xy[(i,j)] = True #set the original to True
            else if (i == 0): #here j will not == 0
                #start with matching z and y
                if (z[j-1] == y[j-1]):
                    hash_xy[(i,j)] = hash_xy[(i, j-1)] #if can be in first prefix
            else if (j ==0): #here i will not == 0
                #empty y
                if (z[i-1] == x[i-1]):
                    hash_xy[(i,j)] = hash_xy[(i-1, j)] #x prefix keep going
            else:
                #here i != 0 and j != 0, 3 cases
                curr_z = z[i + j -1]
                #case1: only x matches to curr_z
                if (x[i-1] == curr_z && y[j-1] != curr_z):
                    hash_xy[(i,j)] = hash_xy[(i-1, j)] #x keep goinf
                #case2: only y matches to curr_z
                else if (y[j-1] == curr_z && x[i-1] != curr_z):
                    hash_xy[(i,j)] = hash_xy[(i, j-1)] #y keep going
                #case3: both x and y matches to curr_z
                else if (y[j-1] == curr_z && x[i-1] == curr_z):
                    hash_xy[(i,j)] = (hash_xy[(i, j-1)] || hash_xy[(i-1, j)])
                #all other cases are still False
    return hash_xy[(len(x), len(y)]
```

Runtime: Since we are running over x and y in a double for loop, the runtime of my algorithm will be $O(|x| * |y|)$. Inserting, adjusting, or getting values in keys and values in a dictionary takes constant time.