CSE 101 Exam 3 Reference Notes | Professor Daniel Kane
Andrew Onozuka | A16760043 | Section A01
Friday, March 10th, 2023

Exam Topics: **Greedy Algorithm** -Huffman Codes -MSTs **Dynamic Programming** -Basic Concepts -LCSS -Knapsack -CMM -All-Pairs Shortest Path -MIS in Trees -Traveling Salesman

Prefix Free Encodings | **Definition**: an encoding is prefix-free if the encoding of no letter is a prefix of the encoding of any other. **Optimal Encoding** | **Problem**: given a string S, find a prefix-free encoding that encodes S using the fewest number of bits.

We make a huffman tree with the lowest frequency letters at the lowest depths. No matter what the tree structure, two of the deepest leaves are siblings. We can assume it is filled by two least frequent elements.

```
HuffmanTree(L)
  Priority queue Q                          } O(n log(n))
  Insert all elements of L to Q             }
  While(|Q| ≥ 2)                            } O(n) Iterations
    x ← Q.DeleteMin()                       }
    y ← Q.DeleteMin()                       } O(log(n))
    Create z, f(z) = f(x) + f(y)            }
    x and y children of z                   }
    Q.Insert(z)                             }
  Return Q.DeleteMin()
```

Runtime: O(n log(n))

Definition: A tree is a connected graph, with no cycles.

A spanning tree in a graph G, is a subset of the edges of G that connect all vertices and have no cycles.

If G has weights, a minimum spanning tree is a spanning tree whose total weight is as small as possible.

Lemma: For an undirected graph G, any two of the below imply the third:

1. |E| = |V|-1
2. G is connected
3. G has no cycles

Corollary: If G is a tree, then |E| = |V|-1.

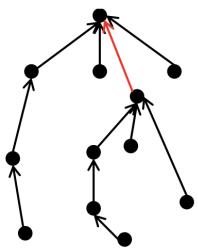
Problem: Given a weighted, undirected graph G, find a spanning tree of G with the lowest possible weight.

```
Kruskal(G)
  Sort edges by weight
  T ← {}
  Create Union Find
  For v ∈ V, New(v)
  For (v,w) ∈ E in increasing order
    If Rep(v) ≠ Rep(w)
      Add (v,w) to T
      Join(v,w)
  Return T
```

Runtime:O(|E|log|E|)

Union Find Implementation

Basic Idea: Each set is a rooted tree with edges pointing towards the representative.



New: Create new node – O(1)
Rep: Follow pointers to root
- O(depth)

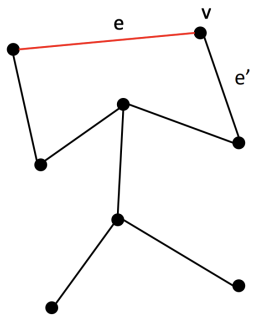
Join: Have one Rep point to other.
- O(depth)

Always connect shallow to deep

Proposition: In a graph G, with vertex v, let e be an edge of lightest weight adjacent to v. Then there exists an MST of G containing e. Furthermore, if e is the unique lightest edge, then *all* MSTs contain e.

- Consider tree T not containing e.
- With extra edge, no longer a tree, must contain a cycle.
- Remove other edge e' adjacent to v from cycle to get T'.
- |T'|=|V|-1, and connected, so T' is a tree.
- wt(T') = wt(T)+wt(e)-wt(e')
  ≤ wt(T)

(because wt(e) is minimal).




```
Prim(G,w)
  Pick vertex s                  \\ doesn't matter which
  For v ∈ V, b(v) ← ∞            \\ lightest edge into v
  T ← {}, b(s) ← 0
  Priority Queue Q, add all v with key=b(v)
  While(Q not empty)
    u ← DeleteMin(Q)
    If u ≠ s, add (u,Prev(u)) to T
    For (u,v) ∈ E
      If w(u,v) < b(v)
        b(v) ← w(u,v)
        Prev(v) ← u
        DecreaseKey(v)
  Return T
```

Runtime: O(|V|log|V| + |E|)
Slightly better than Kruskal

Instead of checking all edges like in Kruskal, Prim's checks just edges from v. Add lightest edge that connects v to a new v

At any stage, have some set S of vertices connected to s. Find cheapest edge connecting S to S^C.

Proposition: In a graph G, with a cut C, let e be an edge of lightest weight crossing C. Then there exists an MST of G containing e. Furthermore, if e is the unique lightest edge, then *all* MSTs contain e.

On the other hand, the longest common subsequence must come from one of these cases. In particular, it will always be the one that gives the biggest result.

LCSS($A_1A_2…A_n$, $B_1B_2…B_m$) =
Max(LCSS($A_1A_2…A_{n-1}$, $B_1B_2…B_m$),
LCSS($A_1A_2…A_n$, $B_1B_2…B_{m-1}$),
[LCSS($A_1A_2…A_{n-1}$, $B_1B_2…B_{m-1}$)+1])
[where the last option is only allowed if $A_n = B_m$]

Dynamic Programming:

1. break problem into smaller subproblems
2. find recursive formula solving one subproblem in simpler ones
3. tabulate answers and solve all subproblems

```
LCSS(A1A2…An, B1B2…Bm)
  Initialize Array T[0…n, 0…m]
    \\ T[i,j] will store LCSS(A1A2…Ai, B1B2…Bj)
  For i = 0 to n
    For j = 0 to m
      If (i = 0) OR (j = 0)
        T[i,j] ← 0
      Else If Ai = Bj
        T[i,j] ← max(T[i-1,j],T[i,j-1],T[i-1,j-1]+1)
      Else
        T[i,j] ← max(T[i-1,j],T[i,j-1])
  Return T[n,m]
```

Notes about DP: general correct proof outline: -prove by induction that each table entry is filled out correctly -use base case and recursion | runtime of dp: -usually [number of subproblems] * [time per] | finding recursion: -often look at first or last choice and see what things look like without that choice | key point: picking right subproblem -enough information stored to allow recursion -not too many

Knapsack Problem:

What is BestValue(C)?

Possibilities:

- No items in bag
  - Value = 0
- Item i in bag
  - Value = BestValue(C-weight(i)) + value(i)

Recursion: BestValue(C) =
Max(0, $\text{Max}_{wt(i) \le C}$ (val(i)+BestValue(C-wt(i))))

```
Knapsack(Wt,Val,Cap)
  Create Array T[0…Cap]
  For C = 0 to Cap          } O(Cap) Subproblems
    T[C] ← 0
    For items i with Wt(i) ≤ C
      If T[C] < Val(i)+T[C-Wt(i)]
        T[C] ← Val(i)+T[C-Wt(i)]
  Return T[Cap]
```

O(#items) time/subproblem

Runtime: O([Cap] [#Items])

$\text{BestValue}_{\le k}$(Cap) = Highest total value of items with total weight at most Cap using only items from the first k.

Base Case: $\text{BestValue}_{\le 0}$(C) = 0

Recursion: $\text{BestValue}_{\le k}$(C) is the maximum of

1. $\text{BestValue}_{\le k-1}$(C)
2. $\text{BestValue}_{\le k-1}$(C- Wt(k))+Val(k)
   [where this is only used if Wt(k) ≤ Cap]

What subproblems do we need to solve?

– We cannot afford to solve all possible CMM problems.

CMM(A_1,…,A_m) requires CMM(A_1,…,A_k) and CMM(A_k,…,A_m).

These require CMM(A_i,A_{i+1},…,A_j), but nothing else.

Only need subproblems C(i,j) = CMM(A_i,A_{i+1},…,A_j) for $1 \le i \le j \le m$.

– Fewer than m^2 total subproblems.

– Critical: Subproblem reuse.

We need to find a recursive formulation.

Often we do this by considering the last step.

For some value of k, last step:
$(A_1A_2…A_k)\cdot(A_{k+1}A_{k+2}…A_m)$

Number of steps:

– CMM(A_1,A_2,…,A_k) to compute first product

– CMM(A_{k+1},…,A_m) to compute second product

– $n_0n_kn_m$ to do final multiply

Recursion CMM(A_1,…,A_m) =
$\min_k$[CMM(A_1,…,A_k)+CMM(A_{k+1},…,A_m)+$n_0n_kn_m$]

Number of Subproblems: One for each $1 \le i \le j \le m$. Total: $O(m^2)$.

Time per Subproblem: Need to check each $i \le k < j$. Each check takes constant time. O(m).

Final Runtime: $O(m^3)$

Base Case: C(i,i) = 0.
(With a single matrix, we don't have to do anything)

Recursive Step:

C(i,j) = $\min_{i \le k < j}$[C(i,k)+C(k+1,j)+$n_in_kn_j$]

Solution order: Solve subproblems with smaller j-i first. This ensures that the recursive calls will already be in your table.

All Pairs Shortest Paths

Problem: Given a graph G with (possibly negative) edge weights, compute the length of the shortest path between every pair of vertices.

Note: `Bellman-Ford` computes single-source shortest paths. Namely, for some fixed vertex s it computes all of the shortest paths lengths d(s,v) for every v.

Repeated `Bellman-Ford`

Easy Algorithm: Run `Bellman-Ford` with source s for each vertex s.

Runtime: $O(|V|^2|E|)$

Dynamic Program

Let $d_k(u,v)$ be the length of the shortest u-v path using at most k edges.

Consider last edge.

Length k-1 path from u to w, edge from w to v.

$d_k(u,v) = \min_w[d_{k-1}(u,w) + \ell(w,v)]$

Algorithm

Runtime: $O(|V|^3 \log|V|)$ $O(|V|^2)$

Base Case:

$$d_1(u,v) = \begin{cases} 0 & \text{if } u = v \\ \ell(u,v) & \text{if } (u,v) \in E \\ \infty & \text{otherwise} \end{cases}$$

Recursion: Given $d_k(u,v)$ for all u, v compute $d_{2k}(u,v)$ using $d_{2k}(u,v) = \min_{w \in V}(d_k(u,w) + d_k(w,v))$.

$O(|V|^3)$

End Condition: Compute $d_1, d_2, d_4, \ldots d_m$ with $m > |V|$.

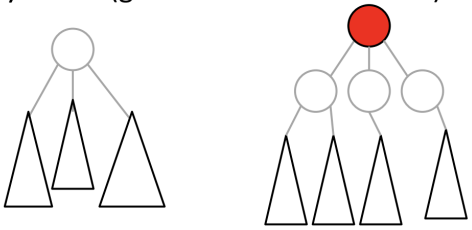
$O(\log|V|)$ iterations

Root not used:

I(G) = Σ I(children's subtrees)

Root is used:

I(G) = 1+Σ I(grandchildren's subtrees)



Algorithm

Runtime: $O(|V|^3)$ $O(|V|^2)$

Base Case:

$$d_0(u,w) = \begin{cases} 0 & \text{if } u = w \\ \ell(u,w) & \text{if } (u,w) \in E \\ \infty & \text{otherwise} \end{cases}$$

Recursion: For each u, w compute: $O(|V|^2)$

$d_k(u,w) = \min(d_{k-1}(u,w), d_{k-1}(u,v_k) + d_{k-1}(v_k,w))$.

End Condition: d(u,w) = d_n(u,w) where n = |V|.

$O(|V|)$ Iterations

Floyd-Warshall Algorithm

Label vertices $v_1, v_2, \ldots, v_n$.

Let $d_k(u,w)$ be the length of the shortest u-w path using only $v_1, v_2, \ldots, v_k$ as intermediate vertices.

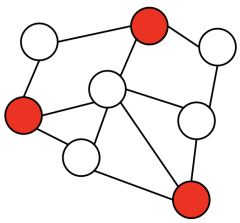
Base Case:

$$d_0(u,w) = \begin{cases} 0 & \text{if } u = w \\ \ell(u,w) & \text{if } (u,w) \in E \\ \infty & \text{otherwise} \end{cases}$$

Independent Set

Definition: In an undirected graph G, an independent set is a subset of the vertices of G, no two of which are connected by an edge.

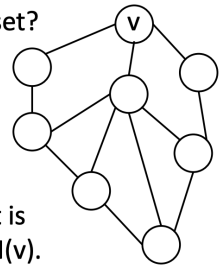
Problem: Given a graph G compute the largest possible size of an independent set of G.



Call answer I(G).

Simple Recursion

Is vertex v in the independent set?



If not: Maximum independent set is an independent set of G-v. I(G) = I(G-v).

If so: Maximum independent set is v plus an independent set of G-N(v). I(G) = 1+I(G-N(v)).

Recursion: I(G) = max(I(G-v), 1+I(G-N(v)))

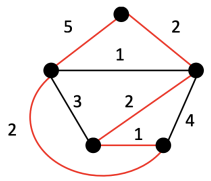
Independent Sets and Components

Lemma: If G has connected components $C_1, C_2, \ldots, C_k$ then

$I(G) = I(C_1)+I(C_2)+\ldots+I(C_k)$.

CSE 101 Exam 3 Reference Notes | Professor Daniel Kane
Andrew Onozuka | A16760043 | Section A01
Friday, March 10th, 2023

Problem: Given a weighted (undirected) graph G with n vertices find a cycle that visits each vertex exactly once whose total weight is as small as possible.



2+1+2+2+5 = 12

$Best_{st,L}(G)$ = Best s-t path that uses exactly the vertices in L.

- Last edge is some $(v,t) \in E$ for some $v \in L$.
- Cost is $Best_{sv,L-t}(G) + \ell(v,t)$.

Full Recursion:

$Best_{st,L}(G) = \min_v[Best_{sv,L-t}(G) + \ell(v,t)]$.

Runtime Analysis

Number of Subproblems:

L can be any subset of vertices (2^n possibilities)
s and t can be any vertices (n^2 possibilities)
$n^2 2^n$ total.

Time per Subproblem:

Need to check every v (O(n) time).

Final Runtime:

$O(n^3 2^n)$
[can improve to $O(n^2 2^n)$ with a bit of thought]

Question 2 (Business Plan I, 35 points). *Harold is starting a new business. He begins with \$0 in capital and wants it to be worth as much as possible at the end of n weeks, at which point he plans to sell. In order to get there, he has k different strategies he can use to make money. Each strategy takes one week to implement, and can be used multiple times as needed. If Harold has x dollars at the start of a week and implements the i^{th} strategy that week, he will have $f_i(x)$ dollars at the end of that week for some increasing function f_i (meaning that $f_i(x) \geq f_i(y)$ whenever $x \geq y$).*

Harold has a simple plan for determining which strategies to use in which order. Each week he will note the number of dollars, x, that he has at the start of the week, and implement the plan with $f_i(x)$ as large as possible. Prove that this strategy gets Harold as much money as possible by the end of the n^{th} week.

We prove by induction on t that this strategy has at least as much money at the end of week t as any other strategy. For a base case we use $t = 0$ and note that all strategies have 0 dollars at the start. Assuming that we know that our strategy has the greatest possible amount of money, x, at the end of week t, we will show that it has the greatest amount of money at the end of week $t+1$. Suppose that Harold's strategy uses the i^{th} strategy on week $t+1$ to end with $f_i(x)$ dollars. Suppose that another schedule ends with y dollars at the end of week t and uses the j^{th} strategy during week $t+1$ to end with $f_j(y)$ dollars. We have that $f_i(x) \geq f_j(x)$ by the method that Harold used to select i. We have that $x \geq y$ by the inductive hypothesis. We therefore, have that $f_j(x) \geq f_j(y)$ since f is increasing. Therefore $f_i(x) \geq f_j(x) \geq f_j(y)$, proving our inductive hypothesis.

Question 3 (Business Plan II, 35 points). *The setup of this problem is similar to that in question 2. Please read that question first.*

Harold later realizes that things are not quite as predictable as he had previously thought. He determines that implementing plan i in a given week has a probability $p_{i,j}$ of earning him j dollars (adding j to his current amount of money) over the course of that week for each integer j between 0 and m. Harold starts with 0 dollars and his goal is to maximize the probability that he will have at least m dollars at the end of n weeks.

Give an algorithm that given n, m and all of the $p_{i,j}$ computes the maximum possible probability with which Harold can achieve this goal. For full credit, your algorithm should run in time $O(nm^2k)$ where k is the number of possible strategies available.

We let $B(d,t)$ be the best probability that Harold can achieve if he has d dollars at the end of week t. We note that $B(d,n)$ is 0 if $d < m$ and 1 otherwise (since at this point, he either has enough money or doesn't). We also know that $B(d,t) = 1$ if $d \geq m$. Otherwise, if Harold has d dollars at the end of week t and uses the i^{th} strategy on week $t+1$ he will have $d+j$ dollars with probability $p_{i,j}$. If he uses the optimal strategy from then onwards, his probability of success will be $B(d+j, t+1)$. Thus, Harold's overall probability of success will be $\sum_{j=1}^{m} p_{i,j} B(d+j, t+1)$.

This means that in order to optimize his probability of success, Harold should use the strategy i that maximizes $\sum_{j=1}^{m} p_{i,j} B(d+j, t+1)$. Thus, $B(d,t) = \max_i(\sum_{j=1}^{m} p_{i,j} B(d+j, t+1))$. This gives rise to the natural dynamic program

```
BestOutcome(n,m,p)
  Create Array B[0..2m,0..n]
  For t = n to 0
    For d = 0 to 2m
      If d >= m
        B[d,t] = 1
      Else if t = n
        B[d,t] = 0
      Else
        B[d,t] = Max{sum_j p_{ij}B[d+j,t+1]}
  Return B[0,0]
```

To show correctness, we prove that each time $B[d,t]$ is assigned a value, it is assigned the correct value of $B(d,t)$. This can be proved by induction using the base case and recursion proved above. We note that $d+j$ is always at most $2m$ and since we assign values for larger values of t first, $B[d+j, t+1]$ will always be assigned before $B[d,t]$ is. Thus, $B[0,0]$ will eventually be assigned the correct probability of success if Harold has no money at the start of week 0.

To analyze runtime, we note that the main loop has $O(nm)$ iterations. Each iteration takes constant time except for the computation of the maximum. Each term in the max is a sum over $O(m)$ things and we need to consider $O(k)$ many possible values of i. Therefore, this step can be done in $O(mk)$ time. Thus, the total runtime is $O(nm^2k)$.

Question 3 (Exam Strategy, 35 points). *Rain is taking a multiple choice test. There are a total of n questions on the test and the test is scored so that Rain gets 4 points for each question answered correctly and 1 point for each question left blank and 0 points for each incorrect answer. In order to pass, Rain needs to score at least T points in total. Rain is not sure about the answers to all of the questions but believes that if he guesses on the i^{th} question that he has a probability p_i of guessing the answer correctly. Furthermore, these probabilities are independent for different questions. Rain wants to maximize his chances of passing by carefully choosing which questions to guess on and which to leave blank.*

Give an algorithm that given T and the p_i's determines the best probability of passing that Rain can achieve. For full credit, your algorithm should run in time $O(n^2)$ or better.

Hint: Assuming that Rain answers exactly k questions, what is his optimal probability of passing?

We note that Rain should answer the k questions with the highest probability of correctness for some k. Letting $P(k,c)$ be the probability that he gets at least c correct answers if he guesses the k best questions we claim that for $k, c > 0$,

$$P(k,c) = P(k-1,c) + q_k[P(k-1,c-1) - P(k-1,c)] \quad (1)$$

where q_k is the probability of guessing the k^{th} highest probability question correctly. This gives the following algorithm:

```
BestProbability(p)
  Let [q_1,q_2,...,q_n] = SortDescending([p_1,p_2,...,p_n])
  Define an array P[0..n][0..n]
  For k = 0 to n
    P[k,0] = 1
  For c = 1 to n
    P[0,c] = 0
  For k = 1 to n
    For c = 1 to n
      P[k,c] = P[k-1,c] + q_k ( P[k-1,c-1] - P[k-1,c] )
  Return max_k P[k,Ceiling((T-n+k)/4)]
```

For the runtime, the sorting step takes $O(n\log(n))$ time. The initialization of $P[k,0]$ and $P[0,c]$ each take $O(n)$. The nested for loop in the middle takes $O(n^2)$ time and the maximum at the end takes $O(n)$ time. So the total runtime is $O(n^2)$.

To show correctness, first we establish that Rain should guess on the k highest probability of success questions for some value of k. To prove this we note that if Rain guesses on some question i but not on some question j with $p_j > p_i$, then he can only improve his chances of success by switching them.

Finally, we need to show that the values assigned to $P[k,c]$ are the correct values of $P(k,c)$ as defined above. If $c = 0$, then the probability should be 1 for all k because Rain is guaranteed to guess at least 0 questions correctly no matter what. On the other hand, if $k = 0$ and $c > 0$, then there is no way that he can get more than 0 correct if he guesses none, so the probability will be 0. We next prove by induction that each other value assigned to $P[k,c]$ is correct. For this, we use the inductive hypothesis to note that $P[k-1,c]$ and $P[k-1,c-1]$ (which have already been assigned) have values $P(k-1,c)$ and $P(k-1,c-1)$, respectively.

To finish the argument, we need to establish Equation (1). This follows from noting that if Rain got at least c questions right, then either he got at least c of the $k-1$ easiest questions right (which happens with probability $P(k-1,c)$) or he got exactly $c-1$ of these right (which happens with probability $P(k-1,c) - P(k-1,c-1)$) and also got the final question right (which happens with probability q_k). combining these, we get Equation (1).

If Rain guesses on the k easiest questions, his probability of getting at least T points is the probability that he got at least $(T-n+k)/4$ of the guessed questions right, or $P(k, \lceil (T-n+k)/4 \rceil)$. The maximum of this over all k is his best probability of success.

Question 2 (Simple Weights MST, 35 points). *Let G be a weighted, undirected graph with edge weights in $\{1, 2, 3, \ldots, k\}$. Give an $O(k(|V| + |E|))$ for computing the weight of a MST on G.*

The algorithm is quite simple

```
Let tot = |V|-1
For w = 1 ... k-1
  Let G_w be the graph G with only the edges of weight at most w
  tot += number of connected components in G_w - 1
Return tot
```

The runtime analysis is similarly easy. Each G_w is computed in linear time as are its connected components. The final runtime is thus $O(k(|V| + |E|))$.

To show correctness note that running Kruskal's algorithm on G to compute the minimum spanning tree, we first add edges of weight 1 until we cannot add any more without creating a cycle. This happens when all of the connected components of G_1 are connected up. Therefore, there are $|V| - |CC(G_1)|$ edges of weight 1 in T. We then add edges of weight 2 until all the connected components of G_2 are completed. Thus, the total number of edges of weights 1 and 2 is $|V| - |CC(G_2)|$. Thus, the number of edges of weight 2 are $|CC(G_1)| - |CC(G_2)|$. Similarly, the number of edges of weight 3 in the minimum spanning tree is $|CC(G_3)| - |CC(G_2)|$ and so on. Thus the weight of the minimum spanning tree is

$$\begin{aligned}
&|V| - |CC(G_1)| + \sum_{w=2}^{k} w(|CC(G_{w-1})| - |CC(G_w)|) \\
&= |V| + |CC(G_1)| + |CC(G_2)| + \ldots + |CC(G_{k-1})| - k|CC(G_k)| \\
&= |V| + |CC(G_1)| + |CC(G_2)| + \ldots + |CC(G_{k-1})| - k \\
&= (|V| - 1) + (|CC(G_1)| - 1) + (|CC(G_2)| - 1) + \ldots + (|CC(G_{k-1})| - 1),
\end{aligned}$$

which is what our algorithm computes.

Alternative Solution: Alternatively, we can proceed using a modification of Prim's algorithm. In particular, our priority queue will only ever need to store numbers in $\{1, 2, 3, \ldots, k, \infty\}$. This can be implemented more efficiently by using an array of linked lists where the i^{th} entry in the array stores a linked list of all the elements in the queue with key equal to i. It is easy to see that this data structure performs insert and decrease key operations in $O(1)$ time and delete min operations in $O(k)$ time. Prim's algorithm needs to perform $|V|$ inserts and delete mins and at most $|E|$ decrease keys. This leads to a final runtime of $O(k|V| + |E|)$.

This in fact can be improved further. If the set of non-empty bins in this array are stored in a Fibonacci Heap, a clever implementation can perform insert and decrease key operations in constant time, and delete min operations in $O(\log(k))$ time. This gives a final runtime of $O(\log(k)|V| + |E|)$.

Question 2 (Heavier than Average, 35 points). *Professor Hayle has a collection of n rocks. He would like to find a rock from his collection whose weight is at least as heavy as the average. He has a scale that given any subset S of the rocks can weigh them to find the total weight of all rocks in S. Show how Hayle can use the scale to find a rock whose weight at least as heavy as the average. For full credit, you solution should use at most $O(\log(n))$ weighings.*

There is a simple divide and conquer algorithm for this problem.

```
HeavierThanAverage(S)
  If |S| = 1
    Return the element of S
  Else
    Split S into nearly equally sized sets S1 and S2
    Weight S1 and S2
    If Weight(S1)/|S1| > Weight(S2)/|S2|
      Return HeavierThanAverage(S1)
    Else
      Return HeavierThanAverage(S2)
```

For the runtime, note that this algorithm makes a constant number of weighings before reducing to a problem of half the size. Thus the runtime satisfies a recurrence $T(n) = T(n/2 + O(1)) + O(1)$ and by the master theorem, $T(n) = O(\log(n))$.

To show correctness we proceed by induction on $|S|$. If $|S| = 1$, then the single element of S has weight equal to the average and so returning it is acceptable. Assuming that our algorithm works for all smaller sized sets, we note that if $\text{Weight}(S1)/|S1| > \text{Weight}(S2)/|S2|$, then it is not hard to see that $\text{Weight}(S1)/|S1| > (\text{Weight}(S1) + \text{Weight}(S2))/(|S1| + |S2|) = \text{Weight}(S)/|S|$. Thus, the average weight of a rock in $S1$ is more than the average weight of a rock in S. Therefore, since by our inductive hypothesis the algorithm in this case returns a rock whose weight is at least the average weight of rocks in $S1$, we return a correct output. Otherwise, we have $\text{Weight}(S2)/|S2| \geq \text{Weight}(S1)/|S1|$, which similarly implies that the average weight of a rock in $S2$ is at least the average weight of a rock in S, and so similarly out answer will be correct.

Question 3 (Tiling, 35 points). *Let R be a subset of a $3 \times n$ grid. Give an algorithm that determines whether or not R can be tiled by 2×1 rectangles. See Figure 1 below for an example. For full credit, your algorithm should run in $O(n)$ time. You may assume that R is presented by a $3 \times n$ array whose (i, j)-entry tells your algorithm whether or not the square (i, j) is in R or not.*

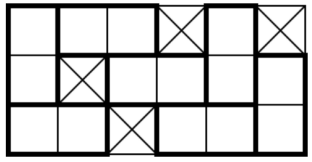


Figure 1: Here R is a 3×6 grid with four squares removed tiled by seven 2×1 rectangles.

We proceed by dynamic programming. We create an $n \times 8$ table ($8 = 2^3$). The entries of the table encode for each column i and each subset of the squares of R in this column whether or not it is possible to tile the rows to the right of i along with the selected squares of row i. We fill in this table starting with large i and working backwards. When $i = n$, we explicitly check to see if the pattern of squares in the last column can be tiled (a finite computation). For smaller values of i, consider all of the (finitely many) was that dominoes could cover the selected tiles in column i. For each of them, see if the columns to the right of $i + 1$ along with the uncovered tiles in column i can be successfully tiled (this will be a lookup in the table). If for any such case, it is possible, mark this entry as possible and otherwise mark it as impossible. Once this table has been filled out, the answer is just a corresponding table entry.

There are $O(n)$ entries to this table, and each is filled out in constant time, so the runtime is $O(n)$.

To show correctness, we note that when filling in an entry we may assume that all previous entries of our table have been correctly filled out. Some configuration is tileable, if and only if there is a way to place our tiles to cover the leftmost column in such a way that the remainder of the region is tileable. Our algorithm checks exactly whether or not this is the case.

Question 2 (Road Trip III, 35 points). *Once again Thomas is travelling along a highway with stops $1, 2, \ldots, n$ in that order trying to get from stop 1 to stop n. This time Thomas has realized that the number of times he needs to stop for gas is less important than the total amount of money that he needs to spend on it. The i^{th} station has a value x_i, the number of miles down the road that it is (with $x_1 < x_2 < \ldots < x_n$), and a price p_i which is the amount charged for enough gas for Thomas to drive a mile. Assume that Thomas starts with no gas at stop 1, but can store an unlimited amount in his tank. Give a polynomial time algorithm to compute the minimum amount that Thomas would need to spend on gas to get to stop n.*

At each stop whose price is cheaper than any previous price, Thomas should pick up exactly enough gas to get to either the end or to the next stop with even cheaper gas (whichever comes first). To prove that this works, imagine that Thomas could take an unlimited amount of gas from any station, putting it into a can labelled with the station name and only had to pay for it when he used it. Clearly Thomas would want to only use gas from the cheapest station he had yet encountered. This is achieved by this solution. An efficient implementation is as follows:

```
Total = 0
MinCost = p_1
for i = 1 to n-1
  Total = Total + MinCost*(x_{i+1}-x_i)
  if p_{i+1} < MinCost
    MinCost = p_{i+1}
return Total
```

The total runtime of this algorithm is clearly $O(n)$.

Question 3 (String Smoothing, 35 points). *Given a string $z_1 z_2 \ldots z_n$ the number of breaks is the number of indices i so that $z_i \neq z_{i+1}$. Given a string $X = x_1 x_2 \ldots x_n$, Ivy would like to alter at most k of the characters to obtain a new string $Y = y_1 y_2 \ldots y_n$ so that the new string has as few breaks as possible. Give a polynomial time algorithm to compute given X and k, the fewest number of breaks in Y that Ivy can achieve.*

We use a dynamic program. For each character C, index i, and integer m we let $FB(C, i, m)$ be the fewest possible breaks attainable by a string that starts with C, and is followed by some string $s_i s_{i+1} \ldots s_n$ where $s_i s_{i+1} \ldots s_n$ is obtained by changing at most m of the characters in $x_i x_{i+1} \ldots x_n$. We claim that:

If $x_i = C$, then $FB(C, i, m) = FB(C, i+1, m)$. This is because in any valid solution, replacing s_i with $x_i = C$ only decreases the number of changes and breaks. Therefore, we can assume that $s_i = C$, and the best we can do is the fewest number of breaks in a string of the form $C s_{i+1} \ldots s_n$.

If $x_i \neq C$, then $FB(C, i, m)$ is the minimum of $FB(x_i, i+1, m) + 1$ and $FB(C, i+1, m-1)$. This is because in any sequence either x_i isn't changed, in which case there is a break C to x_i plus the number of breaks in a string of the form $x_i s_{i+1} \ldots s_n$ with at most m edits (for which the best possible number of breaks is $FB(x_i, i+1, m) + 1$), or x_i is changed. In this case, changing x_i to C gives as few breaks as any other choice, and so we have as many breaks as a string of the form $C s_{i+1} \ldots s_n$ with at most $m-1$ edits (for which the best possible number of breaks is $FB(C, i+1, m-1)$). The best we can do overall is the minimum of these two numbers.

Note that these recursions work when $i = n$ if $FB(C, n+1, m)$ is defined to be 0, since there are no extra breaks after possibly the one between C and s_n.

The algorithm is now as follows:

1. If $k \geq n$ return 0.
2. Let A be an alphabet consisting of all characters appearing in X, plus an additional null character $\emptyset$.
3. Let F be an array with indices $F(C, i, m)$ with $C \in A$ and $1 \leq i \leq n+1$ and $0 \leq m \leq k$.
4. For each C, m, let $F[C, n+1, m] = 0$.
5. For i decreasing from n to 1
   - For each C, m let

$$F[C, i, m] = \begin{cases} F[C, i+1, m] & \text{if } C = x_i \\ \min(F[x_i, i+1, m] + 1, F[C, i+1, m-1]) & \text{else} \end{cases}$$

6. Return $F[\emptyset, 1, k] - 1$

The runtime of this algorithm is dominated by the for loop. It has $O(|A|nk)$ iterations each taking constant time. Therefore, the final runtime is $O(n^2 k)$.

To show correctness, we note by induction that each entry filled in to $F[C, i, m]$ is the correct value of $FB(C, i, m)$ (where $FB(C, n+1, m)$ is defined to be 0). To cover the base case, we note that all the entries with $i = n+1$ are assigned correctly. For later values, the entry assigned is the value given by the recurrence above (since by the inductive hypothesis the previous values of F are correct). Therefore, the final entry in $F[\emptyset, i, k]$ is the best possible number of breakpoints in a valid string $\emptyset Y$, which is one more than the best number of breakpoints for just Y.