CS630: Graduate Algorithms: Homework 1

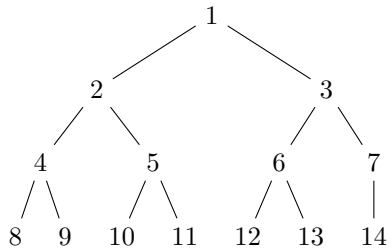
Shivangi, U35642613

September 29, 2023

Exercise 1

Part a

A d-ary heap can be represented in an array like a *binary* heap. Let's try to understand it using an example:



We can represent this above *binary* heap in an array as:
1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14.

We know that children of $heap[1]$ are $heap[2]$ and $heap[3]$. Similarly, children of $heap[2]$ are $heap[4]$ and $heap[5]$. Let's try to generalize this.
So, for a *binary* heap, children of $heap[i]$ are $heap[2i]$ and $heap[2i+1]$.

Similarly, for a d-ary heap,
Children of $heap[0]$ are $heap[1...d]$
Children of $heap[1]$ are $heap[d+1...2d]$
Children of $heap[2]$ are $heap[2d+1...3d]$
Children of $heap[3]$ are $heap[3d+1...4d]$
Children of $heap[i]$ are $heap[id+1...(i+1)d]$

Similarly, the j^{th} child of $heap[i]$ is $heap[id+1+j]$.

Now, let's try to generalize for the parents of $heap[i]$. Using the above formula, we know that 0^{th} child (denoted by J_0) of $heap[i]$ can be given by $id+1$, here $heap[i]$ is the parent. Therefore, after rearranging this, we get:

$Parent = \frac{J_0-1}{d}$

Generalizing it for index i, we get: $Parent = \lfloor \frac{i-1}{d} \rfloor$

Part b

We know that the first level of any kind of heap will have 1 node. We can rewrite it as d^0 nodes in level 1. Similarly, in the subsequent levels, there will be d^1, d^2, d^3... nodes.

We know that there are total n nodes in the heap. So, we can say that: $d^0 + d^1 + d^2 + d^3 ... = n$

After applying the formula for the Sum of the G.P., we can rewrite the above equation as:

$\implies \frac{1-d^{h+1}}{1-d} = n$, where h is the height of the heap.

$\implies 1 - d^(h+1) = n(1-d)$

$\implies d^(h+1) = n(d-1)+1$

After taking $\log_d$ on both sides:

$\implies h+1 = log_d(nd-n+1)$

After solving this equation, we get the expression for the height of $d-ary$ heap as:

$h = \lceil \log_d(n \cdot d - n + 1) - 1 \rceil$

Part c

To perform the EXTRACT-MAX operation, we'll follow these steps:

1. Save the root node in a temporary variable (say, $maxValue$)
2. Replace the root node with the last node of the heap
3. Delete the last node of the heap
4. Heapify the heap to maintain the heap property
5. Return $maxValue$

Let's first discuss the HEAPIFY operation. HEAPIFY operation takes an element at a given index in a heap, ensuring that the subtree rooted at that element satisfies the heap property. Let's first look at the implementation of HEAPIFY operation:

Algorithm 1: HEAPIFY($heap$)

```
  /* heap:  d-ary heap                                                    */
  /* Considering the index starts from 1                                  */
  /* largestChild(heap, i) returns the index of the largest child of node heap[i] */
  /* swap(heap, a, b) swaps the node heap[a] with node heap[b]            */
1 j ← 1;
2 while j ≤ len(heap) do
3     if heap[j] < heap[largestChild(heap, j)] then
4         swap(heap, j, largestChild(heap, j));
5         j ← largestChild(heap, j);
```

Now, let's see the implementation for the EXTRACT-MAX function:

Algorithm 2: EXTRACT-MAX($heap$)

```
  /* heap:  d-ary heap                                                    */
  /* Considering the index starts from 1                                  */
1 if len(heap) = 0 then
2     return None /* The heap is empty                                    */
3 maxValue ← heap[1] ;
4 heap[1] ← heap[len(heap)] ;
5 delete heap[len(heap)] ;
6 HEAPIFY(heap);
7 return maxValue;
```

Now, let's analyze the running time of the EXTRACT-MAX function in terms d and n:
The first three steps of Saving the current rode node in a temporary variable, replacing the root node with the last node of the heap, and deleting the last node will take $O(1)$ time because there are no loops involved and these are direct operations. Next, the HEAPIFY operation will have a time complexity of $O(log_d(n))$ as it needs to compare the current node with its children and potentially swap them. In the worst case, it may need to traverse the height of the tree, which is $log_d(n)$ levels deep.

Overall, the EXTRACT-MAX function in a d-ary max heap has a time complexity of $O(log_d(n))$.

Part d

Following are the steps for the INSERT operation:

1. Insert the new element at the end of the heap
2. Rearrange the heap to maintain the heap property

Let's look at the implementation of the INSERT function:

Algorithm 3: INSERT($heap, elem$)

```
  /* heap: d-ary heap, elem: the element to be inserted                    */
  /* Considering the index starts from 1                                   */
  /* parent(heap, i) returns the index of the parent of node heap[i]       */
  /* swap(heap, a, b) swaps the node heap[a] with node heap[b]             */
1 heap.append(elem);
2 i ← len(heap);
3 while i > 1 do
4     if heap[i] > heap[parent(heap, i)] then
5         swap(heap, i, parent(heap, i));
6         i ← parent(heap, i);
7     else
8         break;
```

Now, let's analyze the running time complexity of the INSERT operation in terms of d and n.
The insertion of the new element at the end of the heap will take $O(1)$ time as it is a simple append operation for dynamic arrays used in languages like Python.
Then, let's discuss the rearrangement of the heap, which is done to maintain the heap property. It consists of a while loop, which basically ensures that the inserted element is moved to the appropriate position, and the heap maintains its heap property. In the worst-case scenario, this loop will have to iterate from the leaf-level to the root. This will take $O(log_d(n)$ iterations.

In conclusion, the INSERT function in a d-ary max heap has a time complexity of $O(log_d(n)$.

Part e

The INCREASE-KEY function is basically used to update a node value in d-ary heap if and only if the current value of that node is less than the new value. It follows the following steps:

1. Check if the new value is greater than the current value of the node. If yes, then proceed to the next step, else flags an error.
2. Set the current value of the node as the new value.
3. Perform a similar rearrange operation as done for the INCREASE-KEY function to maintain the heap property.

Let's look at the implementation of the INCREASE-KEY function:

Algorithm 4: INCREASE-KEY($heap, elem, index$)

```
  /* heap: d-ary heap, elem: the element to be inserted, index: index where the elem needs to be
     inserted                                                              */
  /* Considering the index starts from 1                                   */
  /* parent(heap, i) returns the index of the parent of node heap[i]       */
  /* swap(heap, a, b) swaps the node heap[a] with node heap[b]             */
1 if elem < heap[index] then
2     Flags an error;
3     break;
4 heap[index] ← elem;
5 while index > 1 do
6     if heap[index] > heap[parent(heap, index)] then
7         swap(heap, index, parent(heap, index));
8         index ← parent(heap, index);
9     else
10        break;
```

Now, for the analysis of the time complexity of this algorithm:
Updating the value of $heap[index]$ takes constant time ie. $O(1)$.
The while loop on the other hand is similar to that of the INSERT function and is used to ensure that the updated node is moved up the tree to its appropriate position. And, in the worst case would iterate from the updated node's level(which would be the leaf level in the worst case) to the root. This will take $O(log_d(n)$ iteration.

So, the time complexity of the INCREASE-KEY function is $O)log_d(n)$.

Excercise 2

Part a

Let's consider the following example, where we have two men A and B, and two women 1 and 2. Preferences of these men and women are:
$A : 1, 2$
$B : 2, 1$
$1 : B, A$
$2 : A, B$
After applying the Gale-Shapley algorithm to this Stable Marriage problem. We get the final matches as $A - 1$ and $B - 2$.
In this above case, we can see that there are no instabilities, as both men are partnered with their first choices and there are no circumstances in which they would want to leave their current partners.
We can also see that in this example, while men are paired with their first preferences, both women are not. This leads to the conclusion that **not** every instance of stable matching problem, will have a stable pair (m, w) where both m and n are paired with their first preferences. Hence, the given statement is **False**.

Part b

To prove this statement, let's take an appropriate example. Say, we have three men A, B and C, and three women $1, 2$ and 3 with the following preferences:
$A : 1, 2, 3$
$B : 2, 3, 1$
$C : 2, 1, 3$
$1 : C, B, A$
$2 : B, A, C$
$3 : C, A, B$
We can see that, the pair $(B, 2)$ ranks each other first.
After applying the Gale-Shapley algorithm, we get the following final matches:
$A - 3, B - 2, C - 1$
In this example, the given statement holds **True**. Now let's try to generalize this. Let's consider a problem, where the set of men is M and the set of women is W. Let's assume that, we have a pair $(m, w) \in M$and W respectively, which ranks each other first. But, in the stable matching S, both of them are paired with someone else, (m, w') and (m', w). Although, we know that m prefers w more than w' and hence is unstable. Similarly, w prefers m more than m' and is also unstable. This shows a contradiction of Stable Matching and hence, (m, w) should belong to S.
Hence, the given statement is **True**.

Excercise 3

Let H be the set of hospitals and S be the set of students.
We know that every hospital has some positions which could be more than one and they need to fill all of these positions. Also, the total number of positions from all the hospitals is greater than the total number of students.

Now, let's look at the algorithm:

Algorithm 5: National Resident Matching Algorithm

```
  /* Considering the index starts from 1                                    */
1 Initialize all the students and all the positions in hospitals as unassigned;
2 while there exists an unassigned student who hasn't proposed to every hospital do
3     Select an unassigned student s who hasn't proposed to every hospital;
4     Let h be the highest-ranked hospital in s preference and hasn't been proposed by s;
5     if h has at least 1 unassigned position then
6         s proposed to h;
7         s gets assigned to h;
8     else
9         if h prefers s more than the least preferred student s' assigned to it then
10            Unassign s' from h;
11            Assign s to h;
12        else
13            s remains unassigned;
14 return Assignments
```

Now, let's try to analyze this algorithm and show that it is a stable solution.

Let's assume that there exists a situation where the first type of instability holds ie. s is assigned to h and s' is unassigned but h prefers s' to s.
When a student proposes to a hospital in the algorithm, it implies that it prefers that hospital over any unassigned hospital. Therefore, s would not prefer any unassigned hospital to the one to which it proposed. This eliminates the possibility of s' being assigned to a hospital preferred by s.
If a hospital h receives a proposal from a student s and has unassigned positions, it will accept the proposal. Since h accepted s's proposal, it implies that h prefers s over any unassigned student or any student it has previously rejected.
If h has reached its maximum capacity and s is ranked higher on h's preference list than the least preferred student s' currently assigned to h, the algorithm will unassign s' from h and assign s to h. This ensures that h always prefers the currently assigned students over any unassigned students.
Hence, there won't be a case where s is assigned to h and s' remains unassigned.

Now, let's assume that there exists a situation where the second type of instability occurs, ie. s is assigned to h, and s' is assigned to h'. But s' prefers h over h' and h prefers s' over s.
When a student proposes to a hospital in the algorithm, it implies that it prefers that hospital over any unassigned hospital. If s' is assigned to a different hospital h', it means that s' had the opportunity to propose to h but did not do so because h' was ranked higher on their preference list. Therefore, s' prefer h' to h.
If h has reached its maximum capacity and s is ranked higher on the preference list of h than the least preferred student s' currently assigned to h, the algorithm will unassign s' and assign s to h. This ensures that h always prefers the currently assigned students over any unassigned students.
Similarly, if h' has reached its maximum capacity and s' is ranked higher on its preference list than the least preferred student s currently assigned to h', the algorithm will unassign s from h' and assign s' to h'. This ensures that h' always prefers the currently assigned students over any unassigned students.
Therefore, the algorithm will never return an assignment where the second condition of instability holds.

Excercise 4

Gale-Shapley's algorithm works in a way that ensures that under no circumstances the person can break off the marriage for a better match.
Now, let's explore the truthfulness of this algorithm and see whether lying can lead to better matches or not.
Let's consider an example, where A, B, C are men and $1, 2, 3$ are women and the following are their preferences:
$A : 1, 3, 2$
$B : 3, 1, 2$
$C : 1, 3, 2$
$1 : B, A, C$
$2 : C, B, A$
$3 : A, B, C$
Now, let's say that woman 1 lies about her preferences. Her updated preference is: $1 : B, C, A$

Now, after running the Gale-Shapley algorithm on the truthful preferences, we get the final matches:
$A - 1$
$B - 3$
$C - 2$
Then, I got the following matches after running the algorithm on the preferences where woman 1 lied:
$A - 3$
$B - 1$
$C - 2$

Notice that woman 1 in this case gets her best match. This example proves that lying in some cases can help the woman and get her a better match.

Excercise 5

Part 1.a

The entropy of a random variable X is given by:

$H(X) = -\sum_{x \in X} Pr(x) \log_2(Pr(x))$

Therefore, the expression for Y is given by:

$H(Y) = -\sum_{y \in Y} Pr(y) \log_2(Pr(y))$

Now, we know that f is a bijection, which means that every value in X maps to a unique value in Y, and the same goes for every value in Y. Therefore, we can say that:

$Pr(X) = Pr(Y)$

Therefore, we can rewrite $H(Y)$ as:

$H(Y) = -\sum_{x \in X} Pr(x) \log_2(Pr(x))$

$\implies H(Y) = H(X) = 100 bits$

We can also say that, as Y is a bijection of X, which means that every possible value of X maps to a unique value of Y, and every possible value of Y maps to a unique value of X. This means that all the information is saved and there is no loss of information. And, we know that entropy is the amount of information present in a variable. As both X and Y have the same amount of information, their entropy would be the same too.

Part 1.b

We know that:

$H(X, Y) = H(X) + H(X|Y)$

We also know that conditional entropy is given by:

$H(X|Y) = -\sum_{x \in X} \sum_{y \in Y} Pr(x, y) \log_2(Pr(x|y))$

However, we know that f is a bijection, and therefore, every value in Y maps to exactly one value in X, and vice versa. Therefore, we can write $Pr(x|y)$ as:

$$Pr(x|y) = \begin{cases} 1 & \text{if } y = f(x) \\ 0 & \text{otherwise} \end{cases}$$

So,

$$H(X|Y) = \begin{cases} -\sum_{x \in X} \sum_{y \in Y} Pr(x, y) \log_2(1) & \text{if } y = f(x) \\ -\sum_{x \in X} \sum_{y \in Y} Pr(x, y) \log_2(0) & \text{otherwise} \end{cases}$$

Since $log_2(1)$ is 0 and $log_2(0)$ is $undefined$, therefore:

$H(X,Y) = H(X) = 100bits$

We can also say that, since f is bijection, X and Y are completely determined by each other and hence:

$H(X|Y) = H(Y|X) = 0$

Therefore,

$H(X,Y) = H(X) = 100bits$

Part 1.c

If $g(X)$ is not a bijection and there are at least two values $x1$ and $x2 \in X$ that map to the same value $z \in Z$, then the entropy of Z will generally be less than 100 bits.
The reason for this is that the uncertainty in Z is reduced due to the fact that multiple values of X map to the same value in Z. Basically, if you know that Z is equal to z, it provides less information about the original value of X compared to a situation where the mapping is one-to-one. Therefore, the entropy of Z will be lower than 100 bits.

Part 2.a

The entropy of the variable, based on Samuel Morse's empirical frequency distribution, is approximately 4.22 bits.

```
letter_frequencies = {
    'E': 12000,
    'T': 9000,
    'A': 8000,
    'I': 8000,
    'N': 8000,
    'O': 8000,
    'S': 8000,
    'H': 6400,
    'R': 6200,
    'D': 4400,
    'L': 4000,
    'U': 3400,
    'C': 3000,
    'M': 3000,
    'F': 2500,
    'W': 2000,
    'Y': 2000,
    'G': 1700,
    'P': 1700,
    'B': 1600,
    'V': 1200,
    'K': 800,
    'Q': 500,
    'J': 400,
    'X': 400,
    'Z': 200
}
```

Figure 1: Code for 5.2.a (1)

```python
total_count = sum(letter_frequencies.values())

probabilities = {letter: count / total_count for letter, count in letter_frequencies.items()}

entropy = -sum(prob * math.log2(prob) for prob in probabilities.values())

print(f"The entropy of the variable X is {entropy:.2f} bits.")
```

Figure 2: Code for 5.2.a (2)

```
The entropy of the variable X is 4.22 bits.
```

Figure 3: Code for 5.2.a (3)

Part 2.b

The entropy of the uniform distribution on 26 letters is 4.700 bits.

```python
import math

num_letters = 26

uniform_probability = 1 / num_letters

entropy_uniform = -sum(uniform_probability * math.log2(uniform_probability) for _ in range(num_letters))

print(f"Entropy of Uniform Distribution: {entropy_uniform:.3f} bits")
```

✓ 0.0s

```
Entropy of Uniform Distribution: 4.700 bits
```

Figure 4: Code for 5.2.b

Part 2.c

Morse's goal was to allocate the most straightforward codes, typically shorter ones, to the letters that saw the highest frequency of use in the English language. This strategic assignment of shorter codes to frequently used letters was a deliberate effort to enhance the efficiency of Morse code transmissions. The underlying principle guiding this approach was the pursuit of coding efficiency, where the aim was to expedite communication while minimizing errors. Consequently, Morse's systematic coding approach not only facilitated swifter communication but also reduced the probability of errors occurring during Morse code transmissions, demonstrating his ingenuity and practicality in optimizing this unique communication system.

Part 3.a

Lets prove $H(X,Y) = H(X) + H(Y|X)$

We know that,

$H(X,Y) = -\sum_{x,y \in X,Y} Pr(x,y) \log_2(Pr(x,y))$

And,

$H(Y|X) = \mathbb{E}_x[H(Y|X)]$

$= \sum_{x,y \in X,Y} H(y|x)Pr(x)$

$= -\sum_{x,y \in X,Y} Pr(x)Pr(y|x)\log_2(Pr(y|x))$

We know that, $Pr(x)Pr(y|x) = Pr(x,y)$. Therefore,

$H(Y|X) = -\sum_{x,y \in X,Y} Pr(x,y)\log_2(Pr(y|x))$

Now,

$H(X) = -\sum_{x \in X} Pr(x)\log_2(Pr(x))$

We rewrite $Pr(x)$ as $\sum_{y \in Y} Pr(x,y)$, therefore,

$H(X) = -\sum_{x,y \in X,Y} Pr(x,y)\log_2(Pr(x))$

Now adding $H(X)$ and $H(Y|X)$ will give,

$H(X) + H(Y|X) = -\sum_{x,y \in X,Y} Pr(x,y)\log_2(Pr(x)) + -\sum_{x,y \in X,Y} Pr(x,y)\log_2(Pr(y|x))$

After rearranging,

$\implies -\sum_{x,y \in X,Y} Pr(x,y)\log_2(Pr(x)Pr(y|x))$

$\implies -\sum_{x,y \in X,Y} Pr(x,y)\log_2(Pr(x,y))$

$\implies H(X,Y)$

Part 3.b

We want to prove the property of subadditivity of entropy, it is given by:

$H(X_1, X_2, \ldots, X_n) \leq \sum_{i=1}^{n} H(X_i)$

To prove this, let's first simplify our problem and consider the case where n = 2 ie. we only have two random variables. Now, we want to prove that:

$H(X,Y) \leq H(X) + H(Y)$

Let's assume we have a function $f(x,y)$ such that:

$f(x,y) = -\log_2(Pr(x,y))$

for all $x \in X$ and $y \in Y$

We know that Shearer's Lemma states:

$\mathbb{E}_{(x,y)}[f(X,Y)] \leq \mathbb{E}_x[f(X)] + \mathbb{E}_y[f(Y)]$

Now, let's put in the value of our $f(x,y)$ function to the Shearer's Lemma:

$\mathbb{E}_{(x,y)}[-\log(Pr(x,y))] \leq \mathbb{E}_x[-\log(Pr(x))] + \mathbb{E}_y[-\log(Pr(y))]$

We know that the expected value of a random variable X is defined as:

$\mathbb{E}_{(x)}[g(X)] = \sum_x Pr(x)g(x)$

Let's apply this to our equation:

$-\sum_{x,y \in X,Y} Pr(x,y) \log_2(Pr(x,y)) \leq -\sum_{x \in X} Pr(x) \log_2(Pr(x)) - \sum_{y \in Y} Pr(y) \log_2(Pr(y))$

We know from the definition of entropy that:

$-\sum_{x,y \in X,Y} Pr(x,y) \log_2(Pr(x,y)) = H(X,Y)$

$-\sum_{x \in X} Pr(x) \log_2(Pr(x)) = H(X)$

$-\sum_{y \in Y} Pr(y) \log_2(Pr(y)) = H(Y)$

Putting these values in the equation will give us:

$H(X,Y) \leq H(X) + H(Y)$

Now that we have successfully proven subadditivity for two random variables, let's try to generalize it for n random variables. We want to prove:

$H(X_1, X_2, \ldots, X_n) \leq \sum_{i=1}^{n} H(X_i)$

Now, let's define $f(x_1, x_2, \ldots x_n)$ as:

$f(x_1, x_2, \ldots x_n) = \log_2(Pr(x_1, x_2, \ldots x_n))$

We know the Shearer's Lemma for n random variables is defined as:

$\mathbb{E}_{(X_1,X_2,\ldots,X_n)}[f(X_1, X_2, \ldots, X_n)] \leq \sum_{i=1}^{n} \mathbb{E}_{X_i}[f(X_i)]$

Putting in the value of $f(x_1, x_2, \ldots x_n)$ and applying the expected value definition, we get:

$-\sum_{x_1,x_2,\ldots,x_n} Pr(x_1, x_2, \ldots, x_n) \log_2(Pr(x_1, x_2, \ldots, x_n)) \leq \sum_{i=1}^{n} \left(-\sum_{x_i} Pr(x_i) \log(Pr(x_i))\right)$

This implies that:

$\implies H(X_1, X_2, \ldots, X_n) \leq \sum_{i=1}^{n} H(X_i)$

The intuition behind this inequality is that when we consider the joint entropy of multiple random variables, the uncertainty (entropy) in the joint distribution can't be greater than the sum of the individual uncertainties in the marginal distributions.
In other words, let's say the entropy of a random variable represents the "surprise" associated with it. It's like measuring how much you can be surprised by observing that variable alone. When you combine multiple random variables, you are essentially observing all of them together. The subadditivity principle states that observing all these variables together can't be more surprising than observing each of them individually and adding up their surprises.

Part 3.c

The entropy $H(X)$ when the probability distribution is given by $Pr(X = k) = \frac{1}{2^k} \forall k \in 1, 2, ...$ is given by:

$H(X) = -\sum_{x \in X} Pr(x) \log_2(Pr(x))$

$H(X) = -\sum_{k=1}^{\infty} Pr(X = k) \log_2(Pr(X = k))$

Putting in the values for the probability distribution,

$H(X) = -\sum_{k=1}^{\infty} \frac{1}{2^k} \log_2(\frac{1}{2^k})$

Taking the negative inside,

$H(X) = \sum_{k=1}^{\infty} \frac{1}{2^k} \log_2(2^k)$

Using *log*'s properties,

$H(X) = \sum_{k=1}^{\infty} \frac{1}{2^k} k$

Using the property of G.P.:

$\sum_{n}^{\infty} nr^n = \frac{r}{1-r^2}$

After applying the above formula:

$H(X) = \frac{1}{2}/(1 - \frac{1}{2})^2$

$\implies 2bits$

Exercises 6

Part 1

We know that,

1. The total number of men and women is 4.
2. Preferences of women are fixed.

We can calculate that,

1. Total number of permutations for a single man is 4!.
2. Total number of permutations for all 4 men is $(4!)^4$.

Now let's generalize this for n men and women with similar conditions.

1. Total number of permutations for a single man would be $n!$.
2. Total number of permutations for all n men would be $(n!)^n$.

Part 2

The average number of proposals is 6.60, whereas the maximum number of proposals is 13 and the minimum is 4. This is expected, the average lies between the maximum and minimum.