17-0: **HashMap**

- Arrays allow us to store elements in a list, using ints to reference locations
- ArrayLists give some extra functionality to arrays (automatic resizing, code for inserting, etc)
- Be nice to have a data struture that used Strings (or any arbitrary object) to reference locations
- Conceptually, `soundsMade["cat"] = "meow"`

17-1: **HashMap**

```
HashMap<String, String> hm = new HashMap<String,String>();
```

- Creates a new HashMap – key are Strings, values are Strings
- Can add key / value pairs
- Can get the value associated with a key
- Can check if a key is in the hashmap

17-2: **HashMap**

```
HashMap<String, String> sounds = new HashMap<String, String>();
sounds.put("cat", "meow");
sounds.put("dog", "bark");
sounds.put("cow", "moo");

System.out.println(sounds.get("cat"));
System.out.println(sounds.get("dog"));
System.out.println(sounds.get("cow"));
```

17-3: **HashMap**

- boolean containsKey(Object key)
- boolean containsValue(Object value)
- V get(Object key)
- isEmpty()
- V put(K key, V value)
- V remove(Object key)
- int size()

17-4: **Recursion – Minimum**

- What is a really easy (small!) version of the problem, that I could solve immediately? (Base case)
- How can I make the problem smaller?
- Assuming that I could magically solve the smaller problem, how could I use that solution to solve the original problem (Recursive Case)

17-5: **Recursion – Minimum**

- Write a recursive function that returns the smallest value in the first `size` elements of an array of Comparable objects

- `int minimum(int A[], int size)`

17-6: **Recursion**

```
int minimum(int A[], int size)
{
   if (size == 0)
      return null;
   if (size == 1)
      return A[0];
   int smallest = minimum(A, size - 1);
   if (smallest < A[size - 1])
      return smallest;
   else
      return A[size - 1];
}
```

17-7: **Recursion – Minimum**

- Write a tail-recursive function that returns the smallest value in the first n elements of an array of Comparable objects
- `int minimum(int A[], int size, int smallest)`

17-8: **Recursion**

```
int minimum(int A[], int size, int smallest)
{
   if (size == 0)
      return smallest;
   if (smallest < A[size-1])
      return minimum(A, size - 1, smallest);
   else
      return minimum(A, size - 1, A[n-1]);
}

int minimum(int  A[])
{
   return minimum(A, A.length, Integer.MAX_VALUE);
}
```

17-9: **Problems ...**

- Some of the problems from this lecture are taken from
  - javabat.com
- Really nice way to practice Java programming, check it out!
- Especially good for studying for final!

17-10: **Recursion – Group Sum**

- Input: An array of integers, and a target sum
- Output: true of a subset of the integers add up to the sum, false otherwise
- Examples:

  [3, 5, 7, 11, 13], 15 ==¿ true
  [3, 5, 7, 11, 13], 9 ==¿ false
  [3, 5, 7, 11, 13], 23 ==¿ true
  [3, 5, 7, 11, 13], 40 ==¿ false

17-11: **Recursion – Group Sum**

- Add an extra paramter: Number of elements in array to consider (much like minimum, above) `boolean groupSum(int A[], int size, int target)`

17-12: **Recursion – Group Sum**

```
boolean groupSum(int A[], int size, int target)
{
   if (size == 0)
   {
      return target == 0;
   }
   else if (groupSum(A, size - 1, target))
   {
      return true;
   }
   else
   {
      return groupSum(A, size - 1, target - A[size - 1]);
   }
}
```

17-13: **Recursion – Group Sum**

- Second version: starting index rather than ending index
  - Show on codebat

`int groupSum(int start, int A[], int target)`

17-14: **Recursion – Group Sum**

```
boolean groupSum(int A[], int start, int target)
{
   if (start == A.length)
   {
      return target == 0;
   }
   else if (groupSum(A, start + 1, target))
   {
      return true;
   }
   else
   {
      return groupSum(A, start + 1, target - A[start]);
   }
}
```

17-15: **Recursion – SplitArray**

- Given a list of numbers, can it be split into 2 different sublists that sum to the same value
- See javabat (codebat)

17-16: **Recursion – SplitArray**

```
public boolean splitArray(int[] nums) {
   return splitHelper(nums, nums.length, 0);
}

public boolean splitHelper(int[] nums, int size, int excess)
{
   if (size == 0)
       return excess == 0;
   return (splitHelper(nums, size-1, excess + nums[size-1])) ||
          (splitHelper(nums, size-1, excess - nums[size-1]));
}
```

17-17: **Two player games**

- Board-Splitting Game
  - Two players, V & H
  - V splits the board vertically, selects one half
  - H splits the board horizontally, selects one half
  - V tries to minimize the final value, H tries to maximize the final value

14	5	11	4
12	13	19	7
15	3	10	8
16	1	6	2

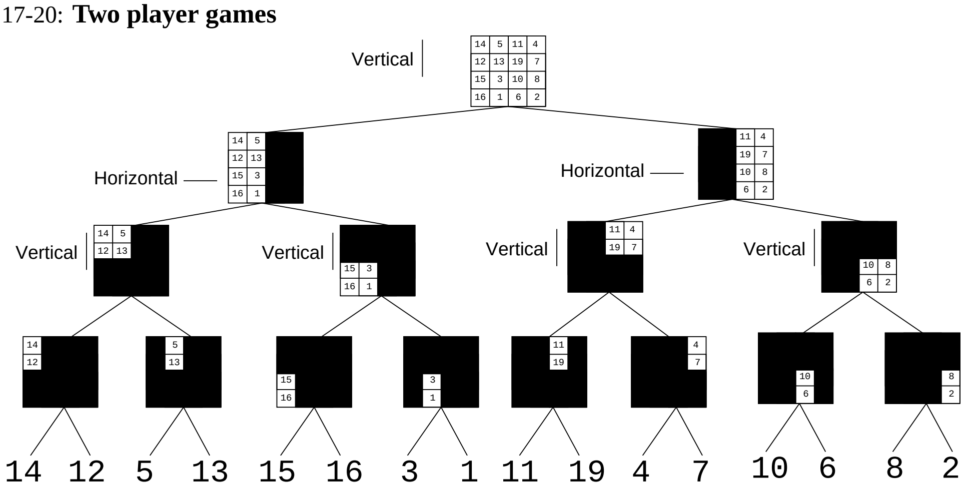
17-18: **Two player games**

- Board-Splitting Game
  - We assume that both players are rational (make the best possible move)
  - How can we determine who will win the game?

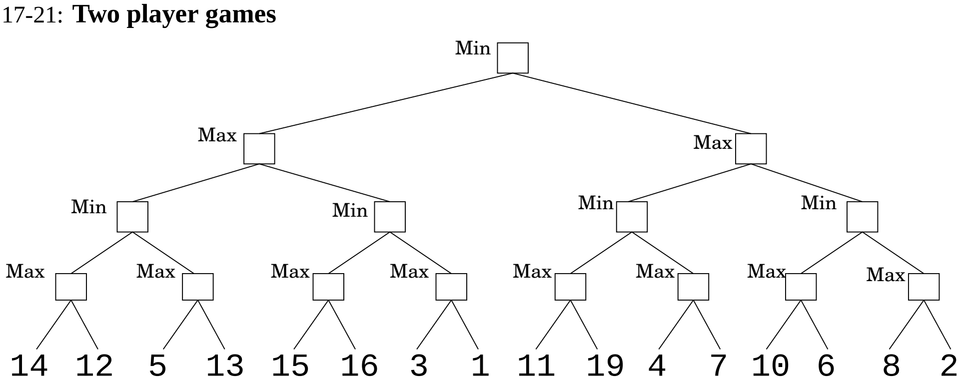
17-19: **Two player games**

- Board-Splitting Game
  - We assume that both players are rational (make the best possible move)
  - How can we determine who will win the game?
    - Examine all possible games!

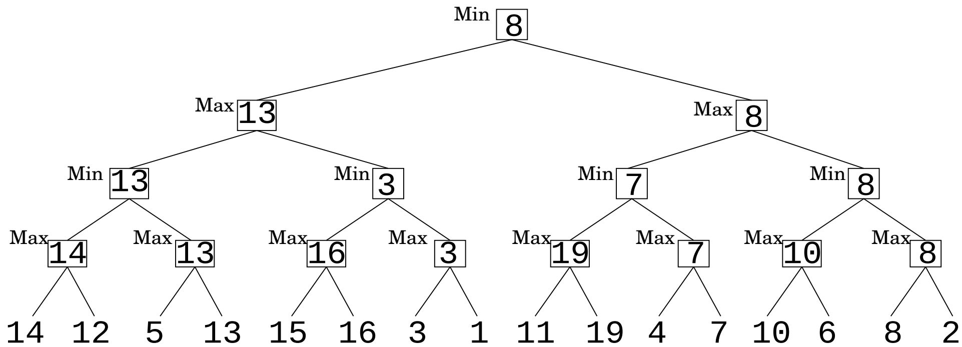
17-20: **Two player games**



17-21: **Two player games**



17-22: **Two player games**



17-23: **Two player games**

- A computer could do this to figure out which move to make
  - Examine all possible moves
  - Examine all possible responses to each move
  - ... all the way to the last move
  - Caclulate the value of each move (assuming opponent plays perfectly)
  -

17-24: **Two player games**

- Could we do this for a real game?
  - Checkers / Chess / Conenct-4 / etc

17-25: **Two player games**

- Could we do this for a real game?
  - Checkers / Chess / Conenct-4 / etc
- No! Too many possible games!

17-26: **Two player games**

- What can we do instead?
  - Create a "board evauluation function"
    - Positive #'s are good for one player, negative #'s good for the other
    - Checkers: # of red pieces - # of black pieces (Can also take position / piece value into account)
  - Search a set number of spaces ahead, use the board evaluation function

17-27: **Two player games**

- Recursion (knew we'd get there eventually ...)
- Write *two* recursive functions
  - `int min(Board B, int level)`

    - Returns the value of the current board, looking `level` moves ahead, assuming that the minimizer goes next
- `int max(Board B, int level)`
    - Returns the value of the current board, looking `level` moves ahead, assuming that the maximizer goes next

17-28: **Two player games**

`int min(Board B, int level)`

- What is the base case?

17-29: **Two player games**

```
int min(Board b, int level)
{
    if (level == 0)
    {
       return b.evalFunction();
    }
    ...
}
```

17-30: **Two player games**

```
int min(Board b, int level)
{
   if (level == 0)
   {
      return b.evalFunction();
   }
   best = Integer.MAX_VALUE;
   for each possible move n we could make
      b.doMove(n);
      moveVal = max(b, n - 1);
      if (moveVal < best)
         best = moveVal;
      b.undoMove(n);
   return best;
}
```

17-31: **Two player games**

```
int max(Board b, int level)
{
   if (level == 0)
   {
      return b.evalFunction();
   }
   best = - Integer.MIN_VALUE;
   for each possible move n we could make
      b.doMove(n)
      moveVal = max(b, n - 1);
      if (moveVal > best)
         best = moveVal;
      b.undoMove(n);
   return best;
}
```

17-32: **Problems ...**

- Go to codingbat.com
- Navigate all java -¿ recursion2
- Do groupSum6, groupSumClump
- If time, look at other problems (splitOdd10 particularly interesting)