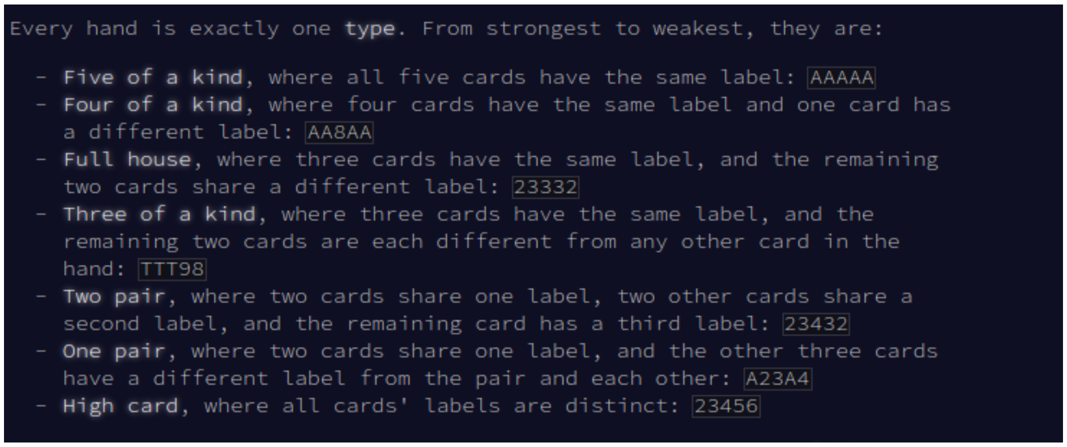
```
In[◦]:= input = "..." ;

In[◦]:= hands = <|"card" → StringTake[#, {1, 5}],
          "bid" → (StringCases[#, RegularExpression["\\s\\d+$"]] // First //
              ToExpression)|> & /@ StringSplit[input, "\n"];

In[◦]:= hands // Short

Out[◦]//Short=
{<|card → T6782, bid → 898|>, <|card → 26T7A, bid → 345|>,
 ≪996≫, <|card → 2J848, bid → 655|>, <|card → 93K65, bid → 966|>}
```

Now we apply 2 layer of parse, first layer will grouping those hands to 7 difference groups base on what kind of cards they have. Second layer will sort them by compare each character. Seem easy, let ‘s see. We have at least 3 functions relative to “Grouping” a collections of element here. SplitBy, GroupBy, GatherBy ΰ . If I checked the document correctly, they nearly the same about input, the difference is SplitBy only apply compare to adj elements, GroupBy produce the result as a association with the key as f[x] (with f is conditional function), values is group of results. Gather will only produce the list of group result.

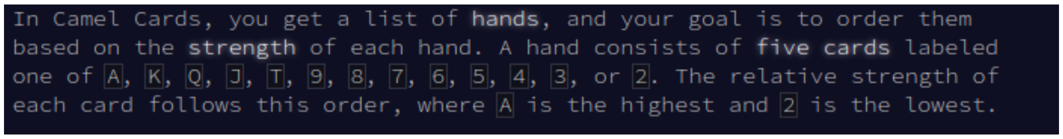


```
Every hand is exactly one type. From strongest to weakest, they are:

  - Five of a kind, where all five cards have the same label: AAAAA
  - Four of a kind, where four cards have the same label and one card has
    a different label: AA8AA
  - Full house, where three cards have the same label, and the remaining
    two cards share a different label: 23332
  - Three of a kind, where three cards have the same label, and the
    remaining two cards are each different from any other card in the
    hand: TTT98
  - Two pair, where two cards share one label, two other cards share a
    second label, and the remaining card has a third label: 23432
  - One pair, where two cards share one label, and the other three cards
    have a different label from the pair and each other: A23A4
  - High card, where all cards' labels are distinct: 23456
```

```
In[∘]:= handGroupingFunction[hand_Association] := Module[{
           c = StringSplit[hand["card"], ""]
          },
          ⎧ 7  (Counts[c] // Values) == {5}
          ⎪ 6  (Counts[c] // Sort // Values) == {1, 4}
          ⎪ 5  (Counts[c] // Sort // Values) == {2, 3}
          ⎨ 4  (Counts[c] // Sort // Values) == {1, 1, 3}
          ⎪ 3  (Counts[c] // Sort // Values) == {1, 2, 2}
          ⎪ 2  (Counts[c] // Sort // Values) == {1, 1, 1, 2}
          ⎩ 1  (Counts[c] // Sort // Values) == {1, 1, 1, 1, 1}
         ]

In[∘]:= handsGroup = SortBy[GroupBy[hands, hand ↦ handGroupingFunction[hand]], Keys[#][[1]] &];

In[∘]:= handsGroup // Keys
Out[∘]= {1, 2, 3, 4, 5, 6, 7}

In[∘]:= MapThread[<|"number of hands" → #1, "type" → #2|> &,
         {Length[handsGroup[#]] & /@ Keys[handsGroup], Keys[handsGroup]} ]

Out[∘]= {<|number of hands → 214, type → 1|>,
         <|number of hands → 260, type → 2|>, <|number of hands → 160, type → 3|>,
         <|number of hands → 175, type → 4|>, <|number of hands → 89, type → 5|>,
         <|number of hands → 101, type → 6|>, <|number of hands → 1, type → 7|>}
```

Now sort each group by character

Oh sorry, so we have a rule here, we not simply comparing by alphabet value of them


In Camel Cards, you get a list of **hands**, and your goal is to order them based on the **strength** of each hand. A hand consists of **five cards** labeled one of `A`, `K`, `Q`, `J`, `T`, `9`, `8`, `7`, `6`, `5`, `4`, `3`, or `2`. The relative strength of each card follows this order, where `A` is the highest and `2` is the lowest.

Another question rose here, what is the definition of bigger, equal and less than, how Sorting actually work . There is no official document to teach me about how to write a comparing function in Wolfram language . If I remember correctly, in Python, a custom comparing function will return {-1, 0, 1} when compare 2 element. But in Wolfram, think is not like this. Base on this document.

- Sort by default orders integers, rational, and approximate real numbers by their numerical values.
- Sort orders complex numbers by their real parts, and in the event of a tie, by the absolute values of their imaginary parts. If a tie persists, they are ordered by their imaginary parts.
- Sort orders symbols by their names, and in the event of a tie, by their contexts.
- Sort usually orders expressions by putting shorter ones first, and then comparing parts in a depth-first manner.
- Sort treats powers and products specially, ordering them to correspond to terms in a polynomial.
- Sort orders strings as in a dictionary, with uppercase versions of letters coming after lowercase ones. Sort places ordinary letters first, followed in order by script, Gothic, double-struck, Greek, and Hebrew. Mathematical operators appear in order of decreasing precedence.
- Sort [*list*, *p*] applies the ordering function *p* to pairs of elements in *list* to determine whether they are in order. The default function *p* is Order.

Wolfram compare thing in the context of numerical, the custom function p, if apply, its job is like a converter, not a compare-one .It didn’t return True or False, 0 or 1. But return a numerical value which used to pick suitable position on the sort result.

```
In[∘]:= convertHandToValue[hand_String] := Module[{
           card = StringSplit[hand, ""],
           cardValues = {"A" → 13, "K" → 12, "Q" → 11, "J" → 10, "T" → 9,
             "9" → 8, "8" → 7, "7" → 6, "6" → 5, "5" → 4, "4" → 3, "3" → 2, "2" → 1}
         },
         cardNumbers = card /. cardValues;
         (#[[1]] * 10^4 + #[[2]] * 10^3 + #[[3]] * 10^2 + #[[4]] * 10^1 + #[[5]]) & @ cardNumbers
        ]
```

```
In[∘]:= SortBy[handsGroup[1], hand ↦ convertHandToValue[hand["card"]]] // Short[#, 10] &
```

```
Out[∘]//Short=
 {<|card → 23Q7A, bid → 878|>, <|card → 25Q47, bid → 739|>, <|card → 26T7A, bid → 345|>,
  <|card → 27T34, bid → 134|>, <|card → 2897K, bid → 321|>, <|card → 2935K, bid → 543|>,
  <|card → 2T476, bid → 90|>, <|card → 2T6Q5, bid → 559|>, <|card → 2J4KA, bid → 43|>,
  <|card → 2J876, bid → 713|>, <|card → 2JT9K, bid → 136|>, <|card → 2Q695, bid → 273|>,
  <|card → 2Q93J, bid → 755|>, <|card → 3298A, bid → 325|>, <|card → 32KJT, bid → 733|>,
  <|card → 2K597, bid → 742|>, ≪182≫, <|card → KAQ25, bid → 927|>,
  <|card → A568Q, bid → 98|>, <|card → A5TQ9, bid → 805|>, <|card → A6K78, bid → 160|>,
  <|card → A75T9, bid → 425|>, <|card → A7Q8T, bid → 983|>, <|card → A7KJT, bid → 309|>,
  <|card → A83K4, bid → 905|>, <|card → A937J, bid → 37|>, <|card → A96Q2, bid → 302|>,
  <|card → A984Q, bid → 821|>, <|card → ATKJ5, bid → 82|>, <|card → AJ482, bid → 255|>,
  <|card → AQ2T6, bid → 99|>, <|card → AK26J, bid → 964|>, <|card → AKT86, bid → 936|>}
```

Nice .

```
In[∘]:=
In[∘]:= handsWithRank = MapThread[<|"card" → #1[["card"]], "bid" → #1[["bid"]], "rank" → #2|> &,
          {Values[SortBy[#, (hand ↦ convertHandToValue[hand["card"]])] & /@ handsGroup ] //
            Flatten, Range[Length[hands]]}]
```

```
Out[∘]= {<|card → 23Q7A, bid → 878, rank → 1|>, <|card → 25Q47, bid → 739, rank → 2|>,
         <|card → 26T7A, bid → 345, rank → 3|>, <|card → 27T34, bid → 134, rank → 4|>,
         <|card → 2897K, bid → 321, rank → 5|>, <|card → 2935K, bid → 543, rank → 6|>,
         <|card → 2T476, bid → 90, rank → 7|>, <|card → 2T6Q5, bid → 559, rank → 8|>,
         <|card → 2J4KA, bid → 43, rank → 9|>, ··· 982 ···, <|card → KKKK7, bid → 773, rank → 992|>,
         <|card → KKAKK, bid → 579, rank → 993|>, <|card → A3AAA, bid → 216, rank → 994|>,
         <|card → AA7AA, bid → 157, rank → 995|>, <|card → AAAA4, bid → 488, rank → 996|>,
         <|card → AAAKA, bid → 673, rank → 997|>, <|card → AAAA9, bid → 387, rank → 998|>,
         <|card → AAAAJ, bid → 738, rank → 999|>, <|card → JJJJJ, bid → 440, rank → 1000|>}
        Full expression not available (original memory size: 0.6 MB)
```

```
In[∘]:= #["rank"] * #["bid"] & /@ handsWithRank // Total
Out[∘]= 247 839 006
```

Wow, wow, wow, surprise, I think everything work perfectly, but the results is wrong. π

It took me 45 minutes just to realize apply the weight of 10^n to each card is wrong. Because it will get corrupted. It just my feeling, I still not sure the mathematic reason about it. Actually, we can only obtain correct result when n >= 13. 13 is the number of type of card from A down to 2. But what is reason that why only >= 13 the right way?

```
In[∘]:= convertHandToValue[hand_String] := Module[{
           card = StringSplit[hand, ""],
           cardValues = {"A" → 13, "K" → 12, "Q" → 11, "J" → 10, "T" → 9,
             "9" → 8, "8" → 7, "7" → 6, "6" → 5, "5" → 4, "4" → 3, "3" → 2, "2" → 1}
         },
         cardNumbers = card /. cardValues;
         (#[[1]] * 13^4 + #[[2]] * 13^3 + #[[3]] * 13^2 + #[[4]] * 13^1 + #[[5]]) & @ cardNumbers
        ]
```

```
In[∘]:= handsWithRank = MapThread[<|"card" → #1[["card"]], "bid" → #1[["bid"]], "rank" → #2|> &,
          {Values[SortBy[#, (hand ↦ convertHandToValue[hand["card"]])] & /@ handsGroup] //
            Flatten, Range[Length[hands]]}]
```

```
Out[∘]= {<|card → 23Q7A, bid → 878, rank → 1|>, <|card → 25Q47, bid → 739, rank → 2|>,
         <|card → 26T7A, bid → 345, rank → 3|>, <|card → 27T34, bid → 134, rank → 4|>,
         <|card → 2897K, bid → 321, rank → 5|>, <|card → 2935K, bid → 543, rank → 6|>,
         <|card → 2T476, bid → 90, rank → 7|>, <|card → 2T6Q5, bid → 559, rank → 8|>,
         <|card → 2J4KA, bid → 43, rank → 9|>, … 982 …, <|card → KKAKK, bid → 579, rank → 992|>,
         <|card → A2AAA, bid → 219, rank → 993|>, <|card → A3AAA, bid → 216, rank → 994|>,
         <|card → AA7AA, bid → 157, rank → 995|>, <|card → AAAKA, bid → 673, rank → 996|>,
         <|card → AAAA4, bid → 488, rank → 997|>, <|card → AAAA9, bid → 387, rank → 998|>,
         <|card → AAAAJ, bid → 738, rank → 999|>, <|card → JJJJJ, bid → 440, rank → 1000|>}
         Full expression not available (original memory size: 0.6 MB)
```

```
In[∘]:= #["rank"] * #["bid"] & /@ handsWithRank // Total
Out[∘]= 247 815 719
```

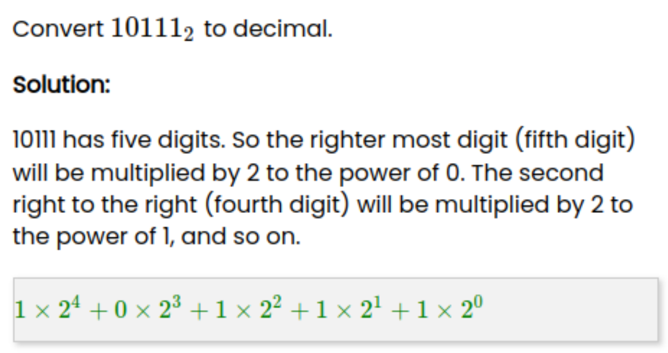
Corrected

I still counted as losing in this problem, because I didn't't understand why >= 13 give me correct result ύ . I will back to check it later

Side note

It took me a while after finish part 2 and go back, try to make my head clear why n >=13 work. From the start, I pick n = 10 and entire function firstNum * 10^4+... +firstNum is simple... unconscious. I just have a feeling that this path will yield the results, my logic is that each will give weight to each card/position. but it simply not enough, why I pick 10. This is the problem, why not pick 2 or anything else. Now I remembered, this pattern did came from the method that convert a binary to decimal form. μ

Example:

In[∘]:=

Convert 10111_2 to decimal.

Solution:

10111 has five digits. So the righter most digit (fifth digit) will be multiplied by 2 to the power of 0. The second right to the right (fourth digit) will be multiplied by 2 to the power of 1, and so on.

$$1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$$

Not hard to understand why we pick n = 13 (I think n > 13 worked is simply random in our case, But n =13 is surely work). Because our problem actually is convert a value in form of 13 digit systems to decimal (10 digits system)

To go deeper a bit, anyone used to learn some 101 course about computer science in university (hum, not me). Or read text book (I actually obtain this experience from the text book, I remember that book name “Code:...”, it here, we will know that human have way to convert the between two type of numbers, binary and decimal. 2 questions will rose here:

1.Why these methods work?
2. More important, why we need these methods?

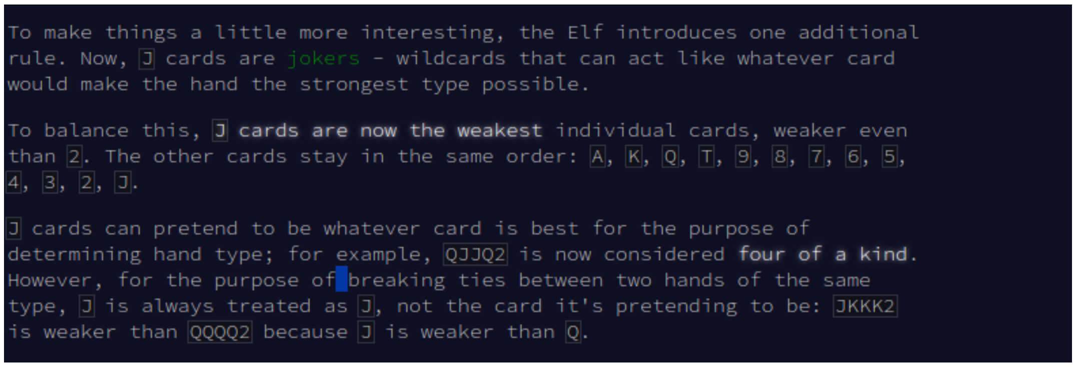
Well, I still not know the answer of the first, so I will focus on the second. Most of the answers will be... because computer using binary, not decimal form of number, ... hum, seem acceptable, but bold, because I am human, why I need to care about computer.

I try to go more generalize, human perspective always recognize the definition of values, even before they invent number system, we always try to give “something” a value on its own. But the problems is, different people, and context, and culture, civilizations will have difference way to “value” everything. That why we need, always need, try to find ways to convert a values from “context A” to a value in “context B”. I think, even the sins humanity have no boundary, at least these method will help us lower the percent of... ah, uh, dumb-ism actions.

In[∘]:=

Part 2

In[∘]:=



To make things a little more interesting, the Elf introduces one additional rule. Now, J cards are jokers - wildcards that can act like whatever card would make the hand the strongest type possible.

To balance this, **J cards are now the weakest** individual cards, weaker even than 2. The other cards stay in the same order: A, K, Q, T, 9, 8, 7, 6, 5, 4, 3, 2, J.

J cards can pretend to be whatever card is best for the purpose of determining hand type; for example, QJJQ2 is now considered **four of a kind**. However, for the purpose of breaking ties between two hands of the same type, J is always treated as J, not the card it's pretending to be: JKKK2 is weaker than QQQQ2 because J is weaker than Q.

µ . god, he know how to make thing more messy. Let we write the rule out here to make clear of our mind. We have 2 new rules:

First, in grouping phase, if J cards exist will “count themself as” other type of card (character) how try make hands that holding them increase to highest rank as much as possible. The only exception as JJJJJ. Like the tree of logic we implement in Part1 will growing more branches

Second, in ranking phase . "J" is itself, and act as lowest value card. This is easier.

Let ‘s implements. I actually rewrote the below code 2 times, change the way to implement the logic from checking number of J first -> split the branch based on the init cards count

```
In[•]:= groupinPhaseFunction[hand_Association] := Module[{
           cards = StringSplit[hand["card"], ""]
         },
         c = Counts[cards];
         j = c["J"];

          ⎧ 7                          (c // Values) == {5}
          ⎪ ⎧ 7  MemberQ[{1, 4}, j]    (c // Sort // Values) == {1, 4}
          ⎪ ⎩ 6  True
          ⎪ ⎧ 7  MemberQ[{2, 3}, j]    (c // Sort // Values) == {2, 3}
          ⎪ ⎩ 5  True
          ⎪ ⎧ 6  MemberQ[{1, 3}, j]    (c // Sort // Values) == {1, 1, 3}
          ⎪ ⎩ 4  True                                                          (*7 xxxxx,
          ⎨ ⎧ 6  MemberQ[{2}, j]
          ⎪ ⎨ 5  MemberQ[{1}, j]       (c // Sort // Values) == {1, 2, 2}
          ⎪ ⎩ 3  True
          ⎪ ⎧ 4  MemberQ[{1, 2}, j]    (c // Sort // Values) == {1, 1, 1, 2}
          ⎪ ⎩ 2  True
          ⎪ ⎧ 2  MemberQ[{1}, j]       (c // Sort // Values) == {1, 1, 1, 1, 1}
          ⎪ ⎩ 1  True
          ⎩ -1                         True

         6 xxxxy, 5 xxxyy, 4 xxxyz, 3 xxyyz , 2 xxyzt, 1 poor hand  *)
        ]

In[•]:= handsGroupWithJoker =
         SortBy[GroupBy[hands, hand ↦ groupinPhaseFunction[hand]] , Keys[#][[1]] &];

In[•]:= convertHandToValueWithJoker[hand_String] := Module[{
           card = StringSplit[hand, ""],
           cardValues = MapThread[#1 → #2 &, {{"A", "K", "Q", "T",
               "9", "8", "7", "6", "5", "4", "3", "2", "J"}, Range[13, 1, -1]}]
         },
         cardNumbers = card /. cardValues;
         (#[[1]] * 13^4 + #[[2]] * 13^3 + #[[3]] * 13^2 + #[[4]] * 13^1 + #[[5]] ) & @ cardNumbers
        ]
```

```
In[◦]:= handsWithRank2 =
         MapThread[<|"card" → #1[["card"]], "bid" → #1[["bid"]], "rank" → #2|> &,
          {Values[SortBy[#, (hand ↦ convertHandToValueWithJoker[hand["card"]])] & /@
               handsGroupWithJoker] // Flatten, Range[Length[hands]]}]

Out[◦]= {<|card → 23Q7A, bid → 878, rank → 1|>, <|card → 25Q47, bid → 739, rank → 2|>,
          <|card → 26T7A, bid → 345, rank → 3|>, <|card → 27T34, bid → 134, rank → 4|>,
          <|card → 2897K, bid → 321, rank → 5|>, <|card → 2935K, bid → 543, rank → 6|>,
          <|card → 2T476, bid → 90, rank → 7|>, <|card → 2T6Q5, bid → 559, rank → 8|>,
          <|card → 2Q695, bid → 273, rank → 9|>, ··· 982 ···, <|card → 99J99, bid → 69, rank → 992|>,
          <|card → TTTJJ, bid → 693, rank → 993|>, <|card → TTTTJ, bid → 399, rank → 994|>,
          <|card → QJQQJ, bid → 918, rank → 995|>, <|card → QJQQQ, bid → 434, rank → 996|>,
          <|card → KJJKK, bid → 611, rank → 997|>, <|card → KJKKK, bid → 190, rank → 998|>,
          <|card → AJJJA, bid → 697, rank → 999|>, <|card → AAAAJ, bid → 738, rank → 1000|>}
        Full expression not available (original memory size: 0.6 MB)

In[◦]:= #["rank"] * #["bid"] & /@ handsWithRank2 // Total

Out[◦]= 248 747 492
```

Scratchpad

```
In[◦]:= SetDirectory["~/nhannht-projects/aoc2023"]

Out[◦]= /home/vermin/nhannht-projects/aoc2023

In[◦]:= Select[<|"B" → 1, "A" → 2, "J" → 2|>, # == Max[<|"B" → 1, "A" → 2, "J" → 2|>] &]

Out[◦]= <|A → 2, J → 2|>

In[◦]:= (KeySelect[<|"A" → 2|>, # == "J" &] // Length) == 0

Out[◦]= True

In[◦]:= <|"B" → 1, "A" → 2, "J" → 2|>

Out[◦]= <|B → 1, A → 2, J → 2|>

In[◦]:= j = <|"B" → 1, "A" → 2|>["J"]

Out[◦]= Missing[KeyAbsent, J]

In[◦]:= MissingQ[j]

Out[◦]= True

In[◦]:= <|"B" → 1, "A" → 2|>["J"]

Out[◦]= Missing[KeyAbsent, J]
```

```
In[∘]:= MissingQ [Missing["KeyAbsent", "J"]]

Out[∘]= True

In[∘]:= <|"B" → 1, "A" → 2, "J" → 2|>[["B"]] = 2

        ••• Set: Association[B → 1, A → 2, J → 2] in the part assignment is not a symbol. ⓘ

Out[∘]= 2

In[∘]:= KeySelect[<|"B" → 1, "A" → 2, "J" → 2|>, # == "J" &]

Out[∘]= <|J → 2|>

In[∘]:= <|"B" → 1, "A" → 2, "J" → 2|> // Keys

Out[∘]= {B, A, J}

In[∘]:=

In[∘]:= inputExample = "32T3K 765
        T55J5 684
        KK677 28
        KTJJT 220
        QQQJA 483";

In[∘]:= handExample = <|"card" → StringTake[#, {1, 5}],
             "bid" →
              (StringCases[#, RegularExpression["\\s\\d+$"]] // First // ToExpression)|> & /@
          StringSplit[inputExample, "\n"]

Out[∘]= {<|card → 32T3K, bid → 765|>, <|card → T55J5, bid → 684|>,
          <|card → KK677, bid → 28|>, <|card → KTJJT, bid → 220|>, <|card → QQQJA, bid → 483|>}

In[∘]:= handsGroupExample =
          SortBy[GroupBy[handExample, hand ↦ handGroupingFunction[hand]], Keys]

Out[∘]= <|2 → {<|card → 32T3K, bid → 765|>},
          3 → {<|card → KK677, bid → 28|>, <|card → KTJJT, bid → 220|>},
          4 → {<|card → T55J5, bid → 684|>, <|card → QQQJA, bid → 483|>}|>

In[∘]:= handsGroupExampleWithJoker =
          SortBy[GroupBy[handExample, hand ↦ groupinPhaseFunction[hand]], Keys]

Out[∘]= <|2 → {<|card → 32T3K, bid → 765|>},
          3 → {<|card → KK677, bid → 28|>}, 6 → {<|card → T55J5, bid → 684|>,
             <|card → KTJJT, bid → 220|>, <|card → QQQJA, bid → 483|>}|>
```

```
In[∘]:= handsWithRankExample =
         MapThread[<|"card" → #1[["card"]], "bid" → #1[["bid"]], "rank" → #2|> &,
          {Values[SortBy[#, (hand ↦ convertHandToValue[hand["card"]])] & /@
              handsGroupExample] // Flatten, Range[Length[handExample]]}]

Out[∘]= {<|card → 32T3K, bid → 765, rank → 1|>,
         <|card → KTJJT, bid → 220, rank → 2|>, <|card → KK677, bid → 28, rank → 3|>,
         <|card → T55J5, bid → 684, rank → 4|>, <|card → QQQJA, bid → 483, rank → 5|>}

In[∘]:= handsWithRankExampleWithJoker =
         MapThread[<|"card" → #1[["card"]], "bid" → #1[["bid"]], "rank" → #2|> &,
          {Values[SortBy[#, (hand ↦ convertHandToValueWithJoker[hand["card"]])] & /@
              handsGroupExampleWithJoker] // Flatten, Range[Length[handExample]]}]

Out[∘]= {<|card → 32T3K, bid → 765, rank → 1|>,
         <|card → KK677, bid → 28, rank → 2|>, <|card → T55J5, bid → 684, rank → 3|>,
         <|card → QQQJA, bid → 483, rank → 4|>, <|card → KTJJT, bid → 220, rank → 5|>}

In[∘]:= #["rank"] * #["bid"] & /@ handsWithRankExample // Total

Out[∘]= 6440

In[∘]:= #["rank"] * #["bid"] & /@ handsWithRankExampleWithJoker // Total

Out[∘]= 5905
```