Definition of Neighborhoods

Small take item with **lowest value-weight ratio** out of the knapsack, simulate filling the knapsack greedily (first take items with highest value-weight ratio), save value of knapsack. Perform best or first swap/fill.

Large for **every** item in the knapsack, take out the item, simulate filling the knapsack greedily, save value of knapsack. Do best or first swap/fill.

Results

Table: Knapsack with N=133

		Value	Time	Iter	Time/Iter
first	**Small**	116493	7.72	4.36	1.94
	Large	119681	123.66	5.12	23.25
best	**Small**	115699	28.264	4.56	6.13
	Large	118287	148.15	4.94	29.15

Mean over 1000 runs

```
knapsack = fill_random()
gesT1 = time.time()
tmp_IterTimes = []
knap = np.zeros(1)
iters = 0
while not np.all(knapsack == knap):
    knap = deepcopy(knapsack)
    candidates = []
    iters += 1
    iterT1 = time.time()
    highestVal = calculate_value(contents(knapsack))
    nr = len(contents(knapsack)) # nr = 1 if SmallNeighbor else len(contents(knapsack))
    for j in range(nr):
        knapsack = deepcopy(knap)
        highestIte = original = get_lowest_ratio(contents(knapsack), j) #nimmt das j-kleinste (bei SmallNeighbor nur das eine)
        knapsack[original] = 0 #take item with lowest value-weight ratio out of the knapsack
        for i in range(len(names)):
            if calculate_weight(contents(knapsack) + [ratio_order[i]]) <= MAX_SIZE and knapsack[ratio_order[i]] == 0:
                if calculate_value(contents(knapsack) + [ratio_order[i]]) > highestVal:
                    highestVal = calculate_value(contents(knapsack) + [ratio_order[i]])
                    highestIte = ratio_order[i]
                    if FirstChoice:
                        knapsack[highestIte] = 1
                        break
                knapsack[highestIte] = 1
        candidates.append((knapsack, calculate_value(contents(knapsack))))
        if SmallNeighbor:
            break
    candidates.sort(key=lambda x: x[1], reverse=True)
    knapsack = candidates[0][0]
    iterT2 = time.time()
    tmp_IterTimes.append(iterT2-iterT1)
gesT2 = time.time()
allgesTimes.append((gesT2-gesT1)*1000)
allIterTimes.append(np.mean(tmp_IterTimes)*1000)
alliters.append(iters)
allresults.append(calculate_value(contents(knapsack)))
```