In [52]:
```
num_elems = np.array([10_000, 20_000, 40_000, 80_000, 160_000])
```

In [53]:
```
insertion_sort = np.array([109,321, 1306, 4729, 23506])
```

In [54]:
```
quick_sort = np.array([1, 3, 7, 16, 32])
```

In [55]:
```
cocktail_sort = np.array([264, 1264, 5140, 22659, 88628])
```

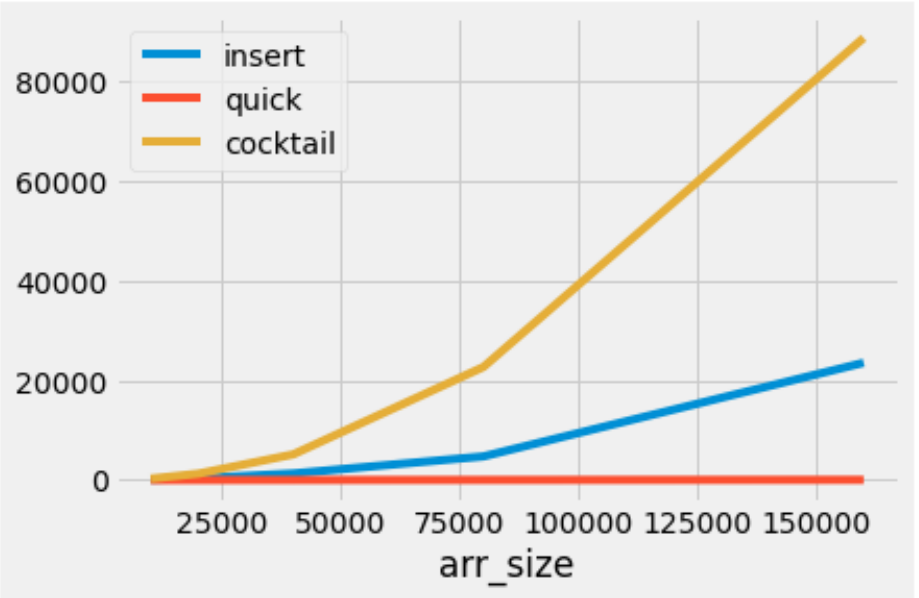
In [56]:
```
data = pd.DataFrame().assign(arr_size = num_elems)
data = data.set_index('arr_size')
data = data.assign(insert = insertion_sort, quick = quick_sort,
                   cocktail = cocktail_sort)

data
```

Out[56]:

arr_size	insert	quick	cocktail
10000	109	1	264
20000	321	3	1264
40000	1306	7	5140
80000	4729	16	22659
160000	23506	32	88628

In [57]:
```
data.plot(kind = 'line')
```

Out[57]: <AxesSubplot:xlabel='arr_size'>



From the graph we can see clear distinctions in performance between the three sorting algorithms, the scaling and fitting of the lines doesn't tell the best picture though. When looking at the tables we can better evaluate the runtimes of each sort. We can clearly observe the O(n log n) time complexity of the quicksort, as the number of elements doubles between iterations we can see that the time increases by roughly a factor of 2 each time. The insertion and cocktail sorts display the average O(n^2) complexity as everytime we double the number of inputs we can observe a square increase in time. When we doubled from 40,000 to 80,000 we saw a 3.6X increase in time for insertion sort, and for the same inputs we saw a 4.4x increase in time for cocktail sort, although it's important to observe that the total amount of time for cocktail sort was about 4 times as long in milliseconds time.

In [58]:
```
cutoff_2 = np.array([281,556,1301,3092,6947])
```

In [59]:
```
cutoff_4 = np.array([279,537,1243,3114,7212])
```

In [60]:
```
cutoff_8 = np.array([263,531,1197,2925,6261])
```

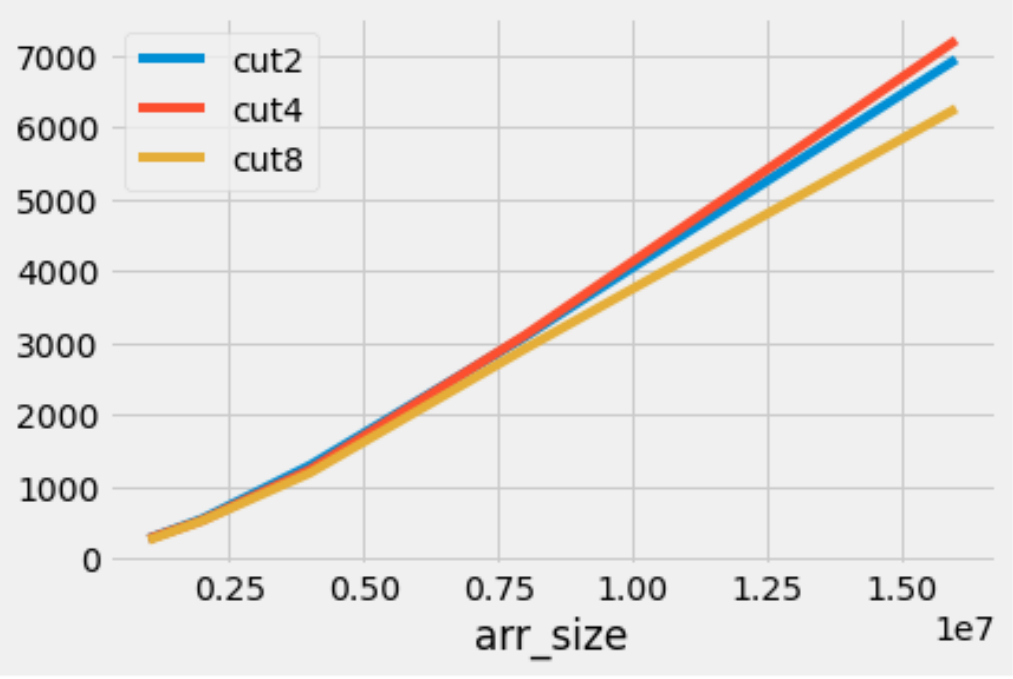
In [62]:
```
data2 = pd.DataFrame().assign(arr_size = num_elems)
data2 = data2.set_index('arr_size')
data2 = data2.assign(cut2 = cutoff_2, cut4 = cutoff_4,
                     cut8 = cutoff_8)

data2
```

Out[62]:

arr_size	cut2	cut4	cut8
1000000	281	279	263
2000000	556	537	531
4000000	1301	1243	1197
8000000	3092	3114	2925
16000000	6947	7212	6261

In [63]:
```
data2.plot(kind = 'line')
```

Out[63]: <AxesSubplot:xlabel='arr_size'>



I should first note that the graph only includes cutoffs 2,4, and 8 because once I got to cutoff 16 the program would never finish even the first iteration on 1,000,000 elements. I think this is because when the cutoff becomes large enough we are calling insertion sort on large subarrays and because of the O(n^2) for each cutoff subarray makes to algorithm way to slow, the runtimes for the larger cutoffs would be even longer because on O(n^2) time complexity would be on an even larger subarray. It appears from the graph that cutoff 8 is the optimal improvement.

In [64]:
```
num_elems = np.array([100_000, 200_000, 300_000, 400_000, 500_000, 600_000])
```

In [65]:
```
quick_sort = np.array([24,39,70,80,96,112])
```

In [66]:
```
modified_quicksort = np.array([23,52,68,83,109,126])
```

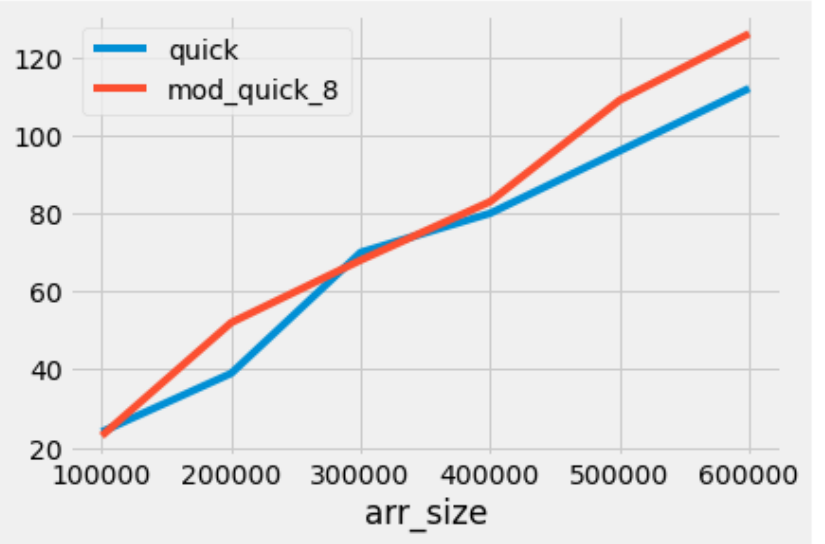
In [70]:
```
data3 = pd.DataFrame().assign(arr_size = num_elems)
data3 = data3.set_index('arr_size')
data3 = data3.assign(quick = quick_sort,
                     mod_quick_8 = modified_quicksort)

data3
```

Out[70]:

arr_size	quick	mod_quick_8
100000	24	23
200000	39	52
300000	70	68
400000	80	83
500000	96	109
600000	112	126

In [71]:
```
data3.plot(kind = 'line')
```

Out[71]: <AxesSubplot:xlabel='arr_size'>



By looking at the graph we can observe that the original quicksort tends to perform better then modified quicksort with optimal cutoff of 8.