ShellSort is a modified version of Insertion sort, where insertion sort is performed multiple times on different gaps on an array. Depending on the order of the gaps, the number of comparisons changes. In case 0 in our testing, the gap is always 1. In case 1, the gap is successive squares of numbers. In case 2, the gap is successive powers of 2 starting at the highest possible and going down to 1. In case 3, the gap is 2^n – 1, ending with 1. In case 3, the gap is 4 \* 2^(k+1) + 3 \* 2^k + 1.

After careful analysis of Shell Sort by generating 100 different sets and taking the average as well as the standard deviation where n = 1000, 3000, 5000, 7000, 9000, 11000, 13000, and 15000,

After a careful analysis of this ShellSort using 100 different randomly generated sets with n=1000, 3000, 5000, 7000, 9000, 11000, 13000, and 15000, it can be concluded that cases 1-4 are significantly better than case 0. Case 0 scales with O(n^2), while the other cases scale with slightly different versions of O(nlogn). This is shown in the following graphical analysis:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **1000** | **3000** | **5000** | **7000** | **9000** | **11000** | **13000** | **15000** |
| Case 0 | 251255.12 | 2256852.34 | 6258949.33 | 12269966.3 | 20271836.7 | 30257664 | 42240477.9 | 56215810.6 |
| Case 1 | 24265.23 | 120326.92 | 255211 | 420089.06 | 608890.16 | 820565.73 | 1051765.75 | 1300942.86 |
| Case 2 | 30352.87 | 150239.54 | 313053.88 | 530536.85 | 784384.38 | 1047946.13 | 1396984.73 | 1664742.96 |
| Case 3 | 27182.35 | 131684.43 | 252797.48 | 426403.64 | 602850.76 | 802768.2 | 1012956.62 | 1290715.7 |
| Case 4 | 35077.89 | 172679.54 | 347861.63 | 577113.77 | 825330.67 | 1128135.89 | 1354083.81 | 1647190.58 |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **3** | **3.47712125** | **3.69897** | **3.84509804** | **3.95424251** | **4.04139269** | **4.11394335** | **4.17609126** |
| Case 0 | **5.40011492** | **6.35350315** | **6.79650144** | **7.08884337** | **7.3068931** | **7.4808354** | **7.62572882** | **7.74985848** |
| Case 1 | **4.38498441** | **5.0803628** | **5.40689939** | **5.62334137** | **5.78453896** | **5.91411338** | **6.02191902** | **6.11425822** |
| Case 2 | **4.48219976** | **5.17678425** | **5.49561909** | **5.72471555** | **5.89452894** | **6.02033896** | **6.14519166** | **6.22134719** |
| Case 3 | **4.434287** | **5.11953443** | **5.40277274** | **5.6298209** | **5.78020981** | **5.90459016** | **6.00559085** | **6.11083059** |
| Case 4 | **4.54503346** | **5.23724088** | **5.54140653** | **5.76126144** | **5.91662798** | **6.05236142** | **6.13164555** | **6.21674385** |

Based on this data, it appears that the fastest of the methods is case 3, but this could be due to statistical error. This method is the fastest for most of the amounts as well as the fastest asymptotically.