**Counting Sort**

1. Faz alocação de memória para realizar a ordenação (Isso permite romper o limite de n.log(n)). Essa alocação consiste em separar os números maior e menor que existem no vetor (por exemplo, 0 e 5). Em seguida, cria-se um vetor de somatório (com posições de 0 a 5). Vale acrescentar que, embora possa ser um vetor de tamanho normal (100 ou 1000 elementos), o fato é que os valores dentro do vetor podem complicar todo o procedimento. Por exemplo: vetor de 3 posições que tem valores de 2 a 5 milhões (o que geraria um vetor auxiliar de 5 milhões de unidades).
2. As chaves precisam ser discretas e idealmente compostas de números inteiros. Por isso, existe uma limitação no seu uso que é a contagem da frequência de números em quantidade limitada (números com virgulas flutuantes são problemáticos – tipo float), bem como a limitação de uso de structs (afinal, na ordenação, não há distinções de outros fatores que senão o inteiro). Pois bem, utilizando-se o vetor alocado, contabiliza-se em cada índice do vetor a quantidade de elementos de existente no vetor desorganizado. Por exemplo: o valor do índice zero, representando o algarismo 0, mostra a quantidade de vezes que o zero deve aparecer no vetor ordenado.
3. Posiciona as chaves corretamente ordenadas. Traduz as informações do vetor auxiliar no vetor ordenado.

Considerando k (quantidade de alocações) e n (contagem de operações), temos que o Counting Sort apresenta os seguintes limites:

1. O(K) o básico e O(n+k) com registros: ou seja, é linear em n, sendo o ideal n>= k. Se for k ~n², o programa passa a apresentar ordem quadrática.
2. Será estável se preservar a ordem relativa dos elementos de entrada.

**Bucket Sort**

É similar ao Counting Sort. Precisa de chaves discretas e com números inteiros e a memória não é constante, uma vez que usa memória extra.

1. Aloca vetor de listas encadeadas (buckets).
2. Preenche buckets de acordo com as chaves.
3. Esvazia buckets, em ordem, inserindo elementos na posição correta.

f(n) = 2k+2n pertence a O(k+n). Por isso, tem a mesma complexidade de tempo do Counting Sort. Porém, em termos de memória, são equivalentes em relação a quantidade de elementos, mas o bucket sort tem um acréscimo por causa dos ponteiros utilizados na memória encadeada. Por isso, m(n,k) = k+ n.E (acaba, no mínimo, dobrando a quantidade de memória por causa dos ponteiros).

**Raddix Sort**

Utiliza como sub rotina o Counting Sort ou o Bucket Sort. Por isso, apresenta as mesmas vantagens e desvantagens dos dois mecanismos.

1. Divide a chave em partes, segundo uma base (8,10, 256). Por exemplo: na base 10, poderíamos utilizar unidades, dezenas e centenas.
2. Para cada base, aplica o Counting Sort ou o Bucket Sort. Isto é, ordenar-se-ia primeiro os números pelas unidades, depois, em um segundo momento, pelas dezenas e, por fim, pelas centenas. Sendo, nesse caso, portanto, necessários três “passadas” pelo vetor.

Como o algoritmo usa o counting sort ou o bucket sort, a base de sua ordenação tem O(n+k), porém será proporicional à quantidade de vezes que os mecanismos de ordenação forem chamados. Por isso, o Raddix Sort tem complexidade de tempo O(d.(n+k)).

O comparativo do Raddix Sort com o Counting Sort mostra que o Counting é mais rápido com pequenas ordenações, que ambos são equivalentes entre vetores de 10 mil a 1 milhão, mas, acima disso, o Raddix começa a apresenta valores significativamente melhores. Isso acontece por causa do operacional do Raddix que passa varias vezes devido a escolha de bases. Mas, conforme o vetor de ordenação fica grande demais, o Counting perde eficiência.