ORDENAÇÃO BASEADA EM INTERPOLAÇÃO LINEAR

Silvio do Lago Pereira[1]
[1] *Prof. Dr. do Departamento de Tecnologia da Informação – FATEC-SP*
slago@fatecsp.br

Resumo

Há duas operações básicas que podem ser usadas para ordenar um vetor: *comparação* e *mapeamento*. Entre os algoritmos que ordenam usando comparação, *Quicksort* é o mais rápido. Por outro lado, entre os algoritmos que ordenam usando mapeamento, *Flashsort* é um dos mais rápidos. Porém, *Flashsort* é um algoritmo híbrido, pois ele também usa comparação. Neste artigo, implementamos um algoritmo de ordenação baseada unicamente em mapeamento, que denominamos *Intersort*, e analisamos empiricamente sua eficiência em relação aos outros dois algoritmos. Os resultados empíricos mostraram que este algoritmo proposto é uma alternativa bem interessante.

1. Introdução

Ordenação é um assunto de grande importância em ciência da computação, tanto teórica quanto prática. Formalmente, dado um vetor v com n itens arbitrários em ordem aleatória, a ordenação consiste em encontrar uma permutação w de v, tal que $w_0 \le w_1 \le w_2 \le \cdots \le w_{n-1}$ [1].

Há duas operações básicas que podem ser usadas para ordenar vetores: *comparação* e *mapeamento*♣. A comparação permite identificar pares de itens que estão fora de ordem, e trocá-los de posição, até que uma permutação ordenada seja obtida. O mapeamento, feito por uma função aplicada a um item, determina a posição em que este item deverá estar na permutação ordenada, sem precisar levar em conta os demais itens do vetor.

Na literatura da área, há vários algoritmos de ordenação que usam apenas comparação como, por exemplo, *Bubblesort*, *Selectionsort*, *Insertionsort*, *Heapsort*, *Mergesort* e *Quicksort* [1, 2, 3]. Por outro lado, *Countingsort* é o único algoritmo de ordenação amplamente conhecido que usa apenas mapeamento [1]. Um resultado provado na literatura [1], e bem conhecido, é que a complexidade temporal dos algoritmos de ordenação que usam apenas comparação é $\Omega(n \lg n)$, enquanto aquela dos algoritmos que usam apenas mapeamento é $\Omega(n)$. Então, claramente, algoritmos baseados apenas em mapeamento são ótimos. Porém, apesar disso, tais algoritmos têm aplicação limitada, podendo ser usados somente para ordenar vetores de itens inteiros, escolhidos dentro de um intervalo bem reduzido. Por este motivo, na prática, os algoritmos que usam mapeamento são híbridos (i.e., usam mapeamento e também comparação) como, por exemplo, *Radixsort*, *Bucketsort* [1] e *Flashsort* [4]. Entre os algoritmos que usam apenas comparação, *Quicksort* é o mais rápido, entre aqueles que são híbridos, *Flashsort* é o mais rápido.

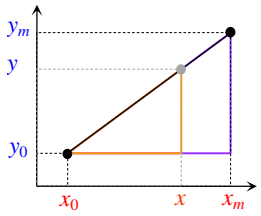
Assim, o objetivo deste artigo é implementar o algoritmo de ordenação *Intersort*, que usa apenas *interpolação linear* (i.e., uma operação de mapeamento) para *mapear* os itens de v às respectivas posições que eles devem ocupar em w, bem como analisar sua eficiência com relação aos algoritmos *Quicksort* e *Flashsort*.

♣ Em inglês, *hashing*.

A organização do restante deste artigo é a seguinte: na Seção 2, mostramos como usar interpolação linear para criar o algoritmo *Intersort*; na Seção 3, apresentamos as versões de *Quicksort* e *Flashsort*, usadas nos experimentos comparativos; na Seção 4, apresentamos a metodologia e os resultados empíricos; e, finalmente, na Seção 5, apresentamos as conclusões finais do trabalho.

2. Ordenação por Interpolação Linear

Interpolação linear [5] é um método numérico para gerar pontos num segmento de reta com extremos (x_0,y_0) e (x_m,y_m). Dada a coordenada x de um ponto nessa reta, sua coordenada y pode ser geometricamente derivada, usando semelhança de triângulos, como mostra a Figura 1.



$$\frac{y-y_0}{y_m-y_0} = \frac{x-x_0}{x_m-x_0}$$

$$\Rightarrow \quad y-y_0 = (y_m-y_0)\cdot\frac{x-x_0}{x_m-x_0}$$

$$\Rightarrow \quad y = \; y_0 + (y_m-y_0)\cdot\frac{x-x_0}{x_m-x_0}$$

Figura 1 – Derivação da fórmula de interpolação linear.

2.1. Mapeamento por Interpolação

Mapeamento [6] é uma função $h(x)$ que transforma um item x num índice $0 \le h(x) < n$. Mais especificamente, mapeamento por interpolação, consiste em interpretar cada item do vetor v como uma coordenada x e usar a sua coordenada y (obtida por interpolação) para determinar sua posição no vetor ordenado w. Neste caso, as abscissas x_0 e x_m são, respectivamente, $\min(v)$ e $\max(v)$ e as ordenadas y_0 e y_m são, respectivamente, 0 e $\text{len}(v)-1$. A fórmula de interpolação linear (Figura 1), adaptada e implementada em *Python* [7], é apresentada na Figura 2.

```
1 from math import floor
2
3 def lif(v):     # linear interpolation function
4     x0 = min(v)
5     xm = max(v)
6     ym = len(v)-1
7     def h(x): return floor(ym*(x-x0)/(xm-x0))
8     return h if x0<xm else None
```

Figura 2 – Síntese da função de interpolação para um vetor v.

A função `lif()` devolve uma função de interpolação linear `h`, adaptada para o vetor `v`. A Figuras 3 mostra o uso da função e Figura 4 exibe o vetor ordenado por ela.

```
>>> v = [53, 68, 10, 42, 37, 91, 86, 25, 74]
>>> h = lif(v)
>>> for x in v: print('h(%s) = %s' % (x,h(x)))
h(53) = 4
h(68) = 5
h(10) = 0
h(42) = 3
h(37) = 2
h(91) = 8
h(86) = 7
h(25) = 1
h(74) = 6
```

Figura 3 – Mapeamento feito pela função de interpolação.

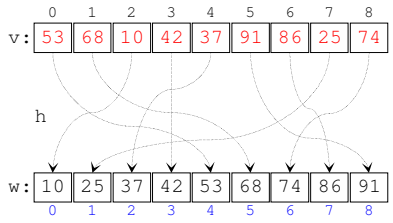


Figura 4 – Funcionamento da ordenação por interpolação.

No melhor caso, a complexidade da ordenação por interpolação linear, tanto temporal quanto espacial, é $O(n)$. O problema é que o melhor caso só ocorre quando o vetor não tem repetição e os itens têm uma distribuição *uniforme* (formam *progressão aritmética aproximada*).

2.2. Tratamento de Colisões

Por usar a função `floor()`, a função $h = \text{lif}(v)$ tem a seguinte propriedade: para quaisquer dois itens $x_i, x_j \in v$, se $x_i \le x_j$, então $h(x_i) \le h(x_j)$. Isso significa que dois itens de v, mesmo que distintos, podem ser mapeados para a mesma posição de w (Figura 5). Quando isso acontece, dizemos que houve uma *colisão*. Para tratar tais colisões, cada posição de w precisaria guardar a lista de itens que colidiram naquela posição, como ilustrado na Figura 6.

```
>>> v = [50, 68, 19, 45, 27, 99, 80, 25, 79]
>>> h = lif(v)
>>> for x in v: print('h(%s) = %s' % (x,h(x)))
h(50) = 3
h(68) = 4
h(19) = 0
h(45) = 2
h(27) = 0
h(99) = 8
h(80) = 6
h(25) = 0
h(79) = 6
```

Figura 5 – Colisões produzidas pela função de mapeamento.

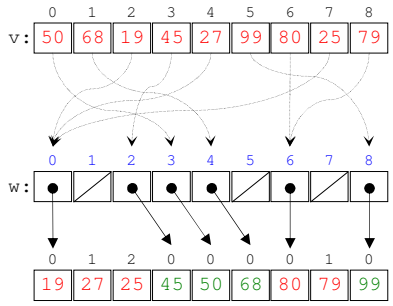


Figura 6 – Esquema para tratamento de colisões.

2.3. O Algoritmo Intersort

O algoritmo recursivo para ordenação por interpolação, criado a partir das ideias apresentadas nas Seções 2.1 e 2.2, é implementado em *Python* na Figura 7.

```
1  def intersort(v):
2      h = lif(v)
3      if not h: return
4      n = len(v)
5      W = [[] for k in range(n)]
6      for x in v:
7          j = h(x)
8          W[j].append(x)
9      i = 0
10     for w in W:
11         if len(w)>1: intersort(w)
12         for x in w:
13             v[i] = x
14             i += 1
```

Figura 7 – O algoritmo *Intersort*.

Nesta função, a linha `2` sintetiza a função `h`, adaptada para o vetor `v` que é dado como entrada. Se `v` tiver apenas um item, ou se todos os seus itens forem iguais, então `x0=min(v)` é igual a `xm=max(v)` e, portanto, `h` é nula (vide linha `9` do algoritmo na Figura 2). Neste caso, a linha `3` é executada e a função termina. Caso contrário, a linha `5` cria uma lista `W` com `n=len(v)` sublistas vazias (para guardar as colisões) e a linha `6` distribui os `n` itens de `v` nas sublistas de `W`. Finalmente, na linha `10`, cada sublista de `W` é ordenada recursivamente (se tiver mais que um item) e seus itens são copiados de volta para `v`. Note que, quando `v` tem apenas dois itens (distintos), a interpolação garante que o menor ficará na sublista `W[0]` e o maior ficará na sublista `W[1]` e, portanto, podemos garantir também que a recursão sempre termina.

No melhor caso, quando os itens de `v` têm distribuição uniforme, a complexidade do algoritmo *Intersort*, tanto temporal quanto espacial, é $O(n)$. Porém, no pior caso, quando a sequência ordenada tem *amplificação fatorial aproximada*, sua complexidade, tanto temporal quanto espacial, é $O(n^2)$. Neste caso, a cada etapa de distribuição dos itens a serem ordenados, a função `h` mapeia o maior item para a sublista `W[n-1]` e todos os demais itens para a sublista `W[0]` (Figura 8).

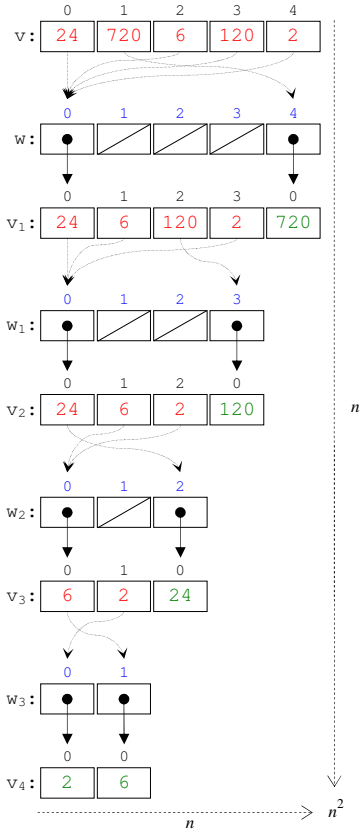


Figura 8 – Pior caso para o algoritmo *Intersort*.

3. Algoritmos de Referência

Para avaliarmos a eficiência do algoritmo *Intersort*, vamos compará-lo aos algoritmos *Quicksort* e *Flashsort*. Como dissemos anteriormente, esses dois algoritmos de ordenação foram escolhidos como referência por serem considerados os *mais eficientes* entre aqueles baseados em comparação e em mapeamento, respectivamente.

3.1. O Algoritmo Quicksort

Quicksort é um algoritmo de ordenação por comparação [3], que usa uma estratégia de *divisão e conquista*. A divisão é feita por *partição* e a conquista por *recursão*.

Dado um vetor $v[start..end]$, a operação de partição *compara* os itens de v a um item $pivot$, permuta esses itens, e devolve um índice cut que corta o vetor v em duas partes, $v[start..cut]$ e $v[cut+1..end]$, tais que $x \le pivot$, para todo $x \in v[start..cut]$, e $x \ge pivot$, para todo $x \in v[cut+1..end]$. Essa operação é implementada em *Python* na Figura 9.

```
1  def partition(v,start,end):
2      pivot = v[(start+end)//2]
3      i = start-1
4      j = end+1
5      while True:
6          i += 1
7          j -= 1
8          while v[i] < pivot: i += 1
9          while v[j] > pivot: j -= 1
10         if i<j: (v[i], v[j]) = (v[j], v[i])
11         else: return j
```

Figura 9 – Operação de partição, usada pelo *Quicksort*.

Após a partição de v, a ordenação de cada uma de suas partes é um problema *independente*. Então, para ordenar v completamente, basta ordenar recursivamente cada uma das partes (similar à ordenação das listas de colisões no *Intersort*). A implementação recursiva do *Quicksort*, em *Python*, é apresentada na Figura 10.

```
1  def quicksort(v):
2      qs(v,0,len(v)-1)
3
4  def qs(v,start,end):
5      if start == end: return
6      cut = partition(v,start,end)
7      qs(v,start,cut)
8      qs(v,cut+1,end)
```

Figura 10 – O algoritmo *Quicksort*.

A função `quicksort()` é apenas um *wrapper* para a função `qs()`, que implementa a estratégia recursiva. O melhor caso para `qs()` acontece quando a operação `partition()` sempre divide o vetor em duas partes do mesmo tamanho. Neste caso, a complexidade temporal do *Quicksort* é $O(n \lg n)$ e sua complexidade espacial é $O(\lg n)$, como mostra a Figura 11.

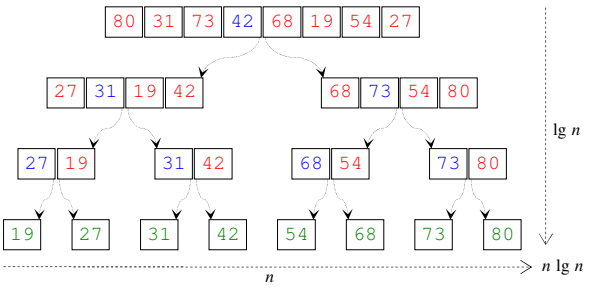


Figura 11 – Melhor caso para o algoritmo *Quicksort*.

O pior caso para `qs()` ocorre quando `partition()` sempre divide o vetor em uma parte com apenas um item e outra parte com todos os demais itens. Neste caso, a complexidade temporal de *Quicksort* é $O(n^2)$ e sua complexidade espacial é $O(n)$, como mostra a Figura 12.

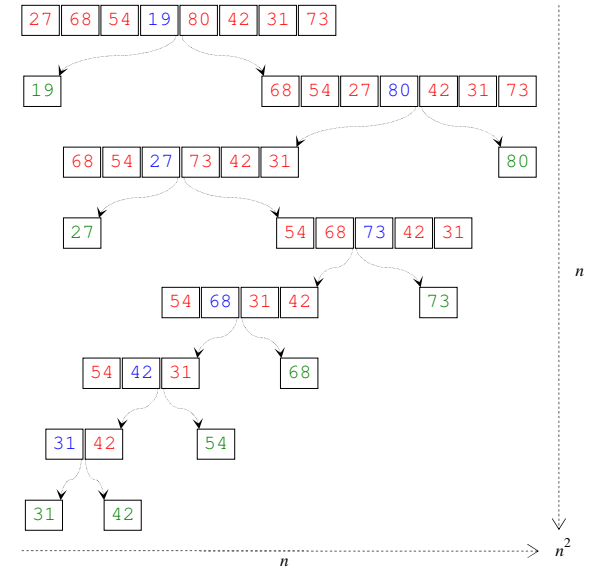


Figura 12 – Pior caso para o algoritmo *Quicksort*.

3.2. O Algoritmo Flashsort

Flashsort é um algoritmo de ordenação cuja operação principal é o mapeamento [4], mas que também usa comparação. O mapeamento reorganiza o vetor de modo que ele fique *parcialmente* ordenado; depois disso, comparações são usadas para garantir sua ordenação *completa*.

Sejam x um item e $v[0..end]$ um prefixo ordenado num vetor v, de tamanho $n>(end+1)$. A operação de inserção insere x em $v[0..end]$, garantindo $v[0..(end+1)]$ ordenado. Essa operação é implementada em *Python* na Figura 13.

```
1  def insert(x,v,end):
2      while end>=0 and v[end]>x:
3          v[end+1] = v[end]
4          end -= 1
5      v[end+1] = x
```

Figura 13 – Operação de inserção, usada pelo *Flashsort*.

De fato, a inserção também é a operação fundamental de um algoritmo de ordenação baseada em comparação chamado *Insertionsort* [2], implementado na Figura 14.

```
1  def insertionsort(v):
2      for i in range(1,len(v)):
3          insert(v[i],v,i-1)
```

Figura 14 – O algoritmo *Insertionsort*.

Note que, quando $x \ge v[end]$, nenhum item precisa ser movido pela função `insert()`. Portanto, no melhor caso, quando o vetor v é crescente, a complexidade temporal do *Insertionsort* é $O(n)$. No pior caso, quando v é decrescente, sua complexidade temporal é $O(n^2)$.

Um fato importante é que, para vetores *parcialmente* ordenados, a complexidade do *Insertionsort* permanece praticamente linear. O *Flashsort* explora justamente esse fato, ou seja, ele ordena o vetor parcialmente e, depois, usa *Insertionsort* para garantir sua ordenação completa.

Para ordenar v parcialmente, o *Flashsort* o particiona em *pilhas* que correspondem às listas de colisões usadas no *Intersort*, usando um vetor de topos t. Inicialmente, t é preenchido com os totais de colisões geradas pelo mapeamento por interpolação; depois, t é transformado numa *soma de prefixos*, resultando num vetor de topos.

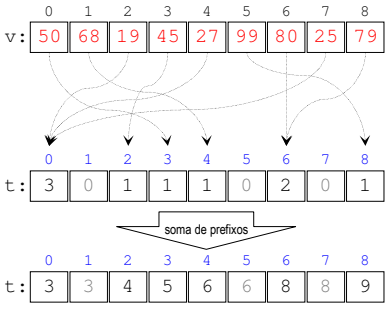


Figura 15 – Síntese do vetor de topos t .

A Figura 15 exemplifica a síntese do vetor t , a partir do vetor v , e a Figura 16 mostra como t particiona v em várias pilhas. Após o particionamento, o *Flashsort* permuta os itens de v para que cada um deles fique na pilha correta, indicada pelo mapeamento por interpolação. No final desta etapa, o vetor já está *quase* ordenado; então, o *Flashsort* usa o *Insertionsort* para concluir a ordenação. A Figura 17 mostra a implementação do *Flashsort*.

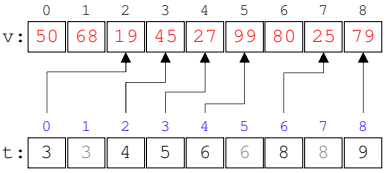


Figura 16 – Partição de v em pilhas, de acordo com t .

```
1  def flashsort(v):
2      h = lif(v)
3      if not h: return
4      n = len(v)
5      t = n*[0]
6      for x in v: t[h(x)] += 1
7      for i in range(1,n): t[i] += t[i-1]
8      m = 0
9      j = 0
10     k = n-1
11     while m < n-1:
12         while j > t[k]-1:
13             j += 1
14             k = h(v[j])
15         while j != t[k]:
16             k = h(v[j])
17             t[k] -= 1
18             (v[j], v[t[k]]) = (v[t[k]], v[j])
19             m += 1
20     insertionsort(v)
```

Figura 17 – O algoritmo *Flashsort*.

Note que a síntese do vetor t (linhas `5` a `7`) e a organização das pilhas induzidas por ele (linhas `11` a `19`) têm complexidade temporal e espacial $O(n)$, pois cada item só pode ser movido uma única vez para a pilha correta. Portanto, no melhor e no pior casos, as complexidades temporais de *Flashsort* são, as mesmas do *Insertionsort* e sua complexidade espacial é $O(n)$. Porém, como o vetor passado como entrada para o *Insertionsort* (linha `20`) está parcialmente ordenado, a complexidade temporal esperada para o *Flashsort* é $O(n)$.

4. Metodologia e Resultados Empíricos

Todos os algoritmos♠ foram implementados com o compilador *Python* 3.4.3, 32 *bits*, rodando numa máquina *Intel(R) Core(TM) i7-5500U* @ 2.40GHz, com 4GB de memória RAM *DDR3*, no sistema operacional *Windows 10*.

♠ Disponível em `www.ime.usp.br/~slago/intersort.py`.

4.1. Dados Usados nos Experimentos

Os vetores usados nos experimentos foram criados com a função `randseq(n,g)`, que recebe um parâmetro `n`, indicando o tamanho do vetor, e um parâmetro opcional `g`, indicando a função geradora de números aleatórios desejada (que pode seguir uma distribuição de probabilidades *uniforme*, *multinomial* ou *gaussiana*). Todas essas funções foram implementadas em *Python*. A Figura 18 mostra exemplos de como os dados são distribuídos nos vetores aleatórios criados nos experimentos, para $n=10^6$.

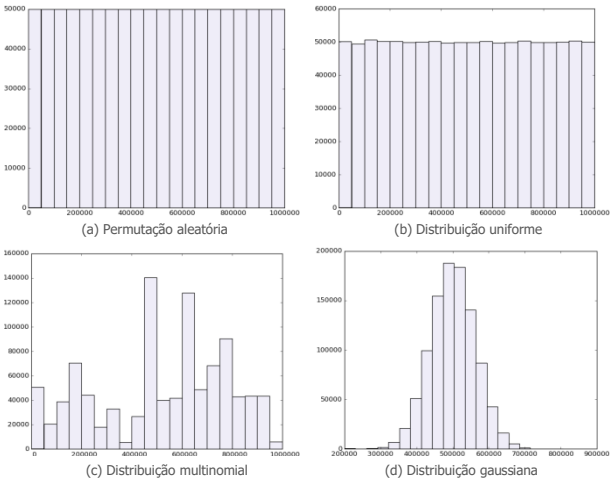


Figura 18 – Distribuições obtidas com as funções geradoras.

A Figura 18-a mostra a distribuição dos dados num vetor criado com a chamada `randseq(n)`, que devolve uma *permutação aleatória* da sequência $[0..n)$; neste caso, claramente, o mapeamento não produz colisões.

A Figura 18-b mostra a distribuição dos dados num vetor criado com `randseq(n,uniform(range(n)))`, que gera uma sequência com n itens escolhidos em $[0..n)$, com distribuição *uniforme* (i.e., todos os itens têm a mesma probabilidade de serem escolhidos). Com esta distribuição, o número médio de colisões em cada posição, geradas pelo mapeamento em 50 vetores, foi 1.4 (*baixo*).

A Figura 18-c mostra a distribuição num vetor criado com `randseq(n,multinomial(range(1000*n)))`, que devolve uma sequência com n itens escolhidos em $[0..1000n)$, com distribuição *multinomial* (i.e., cada item tem uma probabilidade distinta de ser escolhido, definida aleatoriamente *a priori*). Com esta distribuição, o número médio de colisões em cada posição, geradas pelo mapeamento em 50 vetores, foi 2.8 (*moderado*).

A Figura 18-d mostra a distribuição num vetor criado com `randseq(n,gaussian(n,n/2,n/16))`, que devolve uma sequência com n itens escolhidos em $[0..n]$, com distribuição *gaussiana* (com limite n, média $\mu=n/2$ e desvio padrão $\sigma=n/16$). Com esta distribuição, o número médio de colisões em cada posição, geradas pelo mapeamento em 50 vetores, foi 3.6 (*alto*).

Os experimentos foram feitos com vetores de tamanho n variando de 10^5 a 10^6. Os tempos reportados são a média dos tempos medidos para 12 vetores aleatórios, para cada n e cada tipo de distribuição, excluindo-se os tempos mínimo e máximo medidos. Os tempos foram medidos com a função `time()` do *Python*, com precisão de 20 ms. O computador tem 4 núcleos de processamento, sendo um deles para uso exclusivo dos experimentos.

A Tabela I exibe as características das distribuições.

Tabela I – Características das distribuições consideradas.

Distribuição	Permutação	Uniforme	Multinomial	Gaussiana
Colisões	*nenhuma*	*baixo*	*moderado*	*alto*
Repetições	*nenhuma*	*poucas*	*poucas*	*muitas*

4.2. Permutação Aleatória

Os tempos de ordenação de vetores contendo permutações aleatórias são dados na Figura 19. Neste cenário, não há colisões, nem itens repetidos nos vetores.

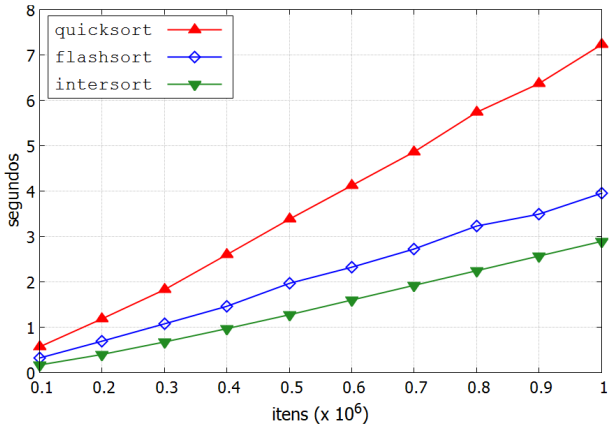


Figura 19 – Permutações aleatórias.

Como esperado, os algoritmos que usam mapeamento foram mais rápidos que *Quicksort*, que usa apenas comparação. Porém, vale ressaltar que esses algoritmos só ordenam vetores numéricos, enquanto *Quicksort* pode ordenar vetores contendo itens de qualquer tipo de dados para o qual seja possível definir uma relação de ordem como, por exemplo, *string*.

Como não há colisões em permutações, *Flashsort* e *Intersort* consomem praticamente a mesma quantidade de memória. Portanto, neste cenário, *Intersort* é mais eficiente que *Flashsort*, pelo menos com relação a tempo.

4.3. Distribuição Uniforme

Os tempos de ordenação de vetores contendo itens com distribuição uniforme são dados na Figura 20. Neste cenário, o número de colisões esperadas é baixo e a quantidade de itens repetidos nos vetores é pequena.

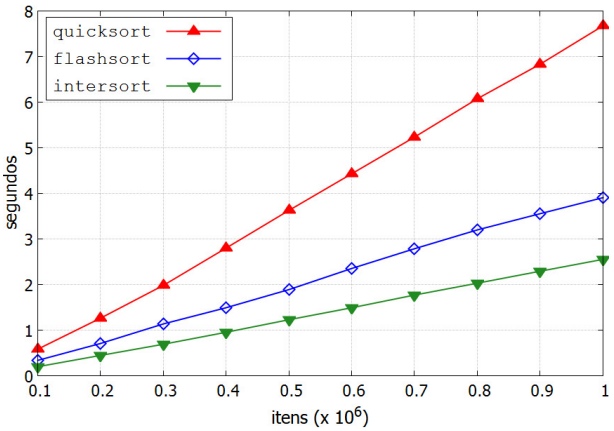


Figura 20 – Itens com distribuição uniforme.

Como o número colisões é baixo, *Intersort* não gasta muito mais memória que *Flashsort*. Ademais, como são escolhidos n itens entre n possibilidades, os itens mapeados para uma mesma lista de colisões são iguais. Isso é vantajoso para o *Intersort*, pois permite que suas chamadas recursivas terminem mais rapidamente. Portanto, neste segundo cenário considerado, *Intersort* continua sendo o algoritmo mais eficiente.

4.4. Distribuição Multinomial

Os tempos de ordenação de vetores contendo itens com distribuição multinomial são dados na Figura 21. Neste cenário, o número de colisões esperadas é moderado e o número de itens repetidos nos vetores é pequeno.

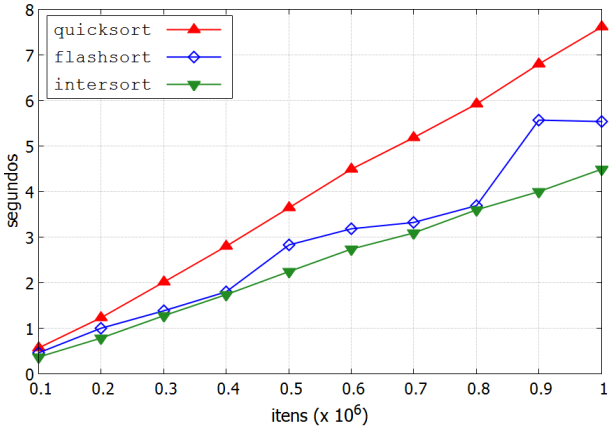


Figura 21 – Itens com distribuição multinomial.

Como o número colisões é moderado, *Intersort* pode gastar mais memória que *Flashsort*. Ademais, como são escolhidos n itens entre $1000n$ possibilidades, na maior parte das listas de colisões, há pouca chance de haver itens iguais. Isso é desvantajoso para *Intersort*, pois exige uma quantidade maior de chamadas recursivas. Mesmo assim, ao contrário do que seria esperado, o *Intersort* foi o algoritmo mais eficiente neste terceiro cenário.

Um ponto interessante observado neste cenário é que, como as probabilidades da distribuição multinomial são escolhidas aleatoriamente para cada novo vetor gerado, o comportamento temporal do *Flashsort* fica um pouco instável (inclusive apresentando picos que superam o tempo do *Quicksort*, quando o tempo máximo medido nos experimentos não é excluído no cálculo do tempo médio reportado, como pode ser visto na Figura 22).

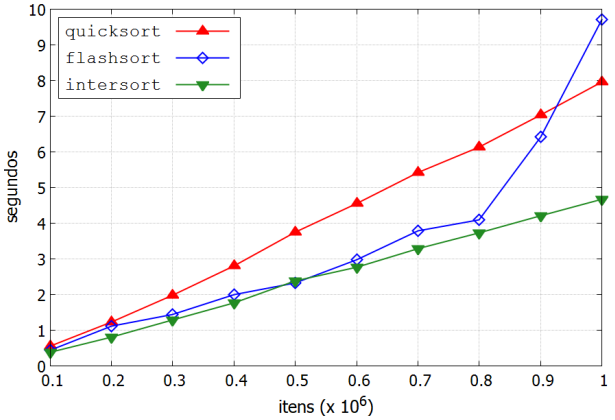


Figura 22 – Comportamento temporal instável do *Flashsort*.

É importante ressaltar que a instabilidade observada na Figura 22 *não* é decorrente da interferência de outros processos em execução simultânea no computador. Como dito anteriormente, os experimentos foram executados em um núcleo dedicado do computador. Além disso, se houvesse alguma interferência, seria totalmente improvável que ela prejudicasse apenas o desempenho do algoritmo *Flashsort* (como se observa no gráfico), pois os algoritmos são executados de forma alternada (como se pode verificar no código-fonte disponível).

4.5. Distribuição Gaussiana

Os tempos de ordenação de vetores contendo itens com distribuição gaussiana são dados na Figura 23. Neste cenário, o número de colisões esperadas é alto e há muitos itens repetidos nos vetores (observe na Figura 18-d que os itens com maior probabilidade de escolha estão concentrados num subintervalo muito estreito de $[0..n]$).

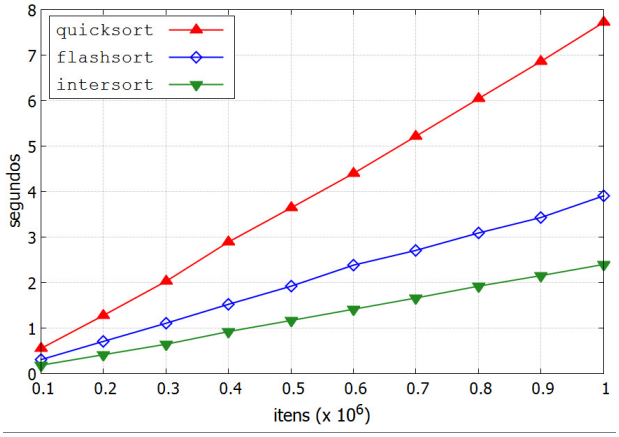


Figura 23 – Itens com distribuição gaussiana.

Como o número colisões é alto, *Intersort* poderia gastar muito mais memória que *Flashsort*; porém, como há muitos itens repetidos nas listas de colisões, assim que eles ficam todos iguais, o *Intersort* pode interromper as chamadas recursivas. Consequentemente, o consumo de memória do *Intersort* não chega a ser tão grande e, além disso, seu consumo de tempo é muito pequeno. De fato, mesmo com um número alto de colisões, entre os quatro cenários considerados, este foi o melhor para o *Intersort*.

5. Conclusões

Os algoritmos de ordenação de vetores descritos na literatura são classificados como algoritmos baseados em *comparação* ou algoritmos baseados em *mapeamento*, dependendo de qual é a principal operação usada por eles para ordenar os itens de um vetor. Aqueles algoritmos baseados em comparação de itens são mais flexíveis, podendo ser usados para ordenar vetores com itens de qualquer tipo para o qual seja possível definir uma relação de ordem como, por exemplo, *string*; porém, o consumo de tempo mínimo destes algoritmos é $\Omega(n \lg n)$. Por outro lado, aqueles algoritmos baseados em mapeamento de itens são mais eficientes, com consumo de tempo mínimo $\Omega(n)$; porém, esses algoritmos são limitados a ordenar apenas vetores cujos itens sejam de tipo numérico.

Entre os algoritmos de ordenação baseada em comparação, o *Quicksort* é considerado um dos mais eficientes, com complexidade de tempo esperada $O(n \lg n)$. Por outro lado, entre os algoritmos de ordenação baseada em mapeamento, *Flashsort* é considerado um dos mais eficientes, com complexidade de tempo esperada $O(n)$. Porém, de fato, *Flashsort* é um algoritmo híbrido, pois ordena vetores usando tanto mapeamento quanto comparação. Assim, o principal objetivo deste artigo foi investigar se um algoritmo baseado *unicamente* em mapeamento, feito por *interpolação linear*, poderia ser uma alternativa mais eficiente para a ordenação de vetores numéricos.

Para verificar essa hipótese, mostramos como implementar em *Python* um algoritmo de ordenação baseada em mapeamento por interpolação linear, que denominamos *Intersort*. Além deste algoritmo, também implementamos os algoritmos *Quicksort*, *Flashsort*, e uma série de algoritmos para geração de vetores contendo números inteiros aleatórios, de acordo com diversas distribuições de probabilidade. Em todos os experimentos comparativos realizados, o *Intersort* foi o algoritmo mais eficiente em todos os cenários considerados (pelo menos com relação ao consumo de tempo, já que o consumo de memória não foi efetivamente medido nos experimentos).

Um fato importante que devemos ressaltar é que, nos cenários avaliados nos experimentos, não usamos vetores contendo sequências de itens com *amplificação fatorial aproximada*, o que faria *Intersort* consumir tempo $O(n^2)$. Tomamos essa decisão pela impossibilidade de gerar sequências desse tipo com 10^6 itens (pois a taxa de crescimento da função fatorial torna impossível representar valores tão grandes na memória do computador). Apesar disso, também ressaltamos que os vetores de inteiros aleatórios usados nos experimentos simulam dados que normalmente são encontrados em situações práticas de ordenação e que, portanto, o algoritmo *Intersort* pode ser uma alternativa viável e eficiente para uso prático.

Referências Bibliográficas

[1] T. H. Cormen et al. **Introduction to Algorithms**, 3rd Edition, MIT Press, Cambridge, 2010.

[2] D. Knuth. **The Art of Computer Programming**, Volume 3: Sorting and Searching, 3rd Edition. Addison-Wesley,1997. Section 5.2.1: *Sorting by Insertion*, pp. 80–105.B.

[3] C. A. R. Hoare. **Quicksort**. The Computer Journal, vol. 5, Issue 1, p. 10-16, 1962.

[4] K. D. Neubert. **The FlashSort Algorithm**. Proceedings of the euroFORTH'97, Oxford, England, 1997.

[5] S. S. Sastry. **Introductory Methods of Numerical Analysis**, 5th Edition, PHI Learning Private Limited, New Delhi, 2012.

[6] I. Müller et al. **Cache-Efficient Aggregation: Hashing is Sorting**, MIT, 2019.

[7] D. Beazley; B. K. Jones. **Python Cookbook**, O'Reilly, USA, 2013.