O Algoritmo de Ordenação Smoothsort Explicado

André L. Pfitzner
andrepfitzner@gmail.com

Paulo E. D. Pinto
pauloedp@ime.uerj.br

Rosa Maria E. M. da Costa
rcosta@ime.uerj.br

Instituto de Matemática e Estatística - Universidade do Estado do Rio de Janeiro
Rua São Francisco Xavier, 524 / sala 6020-B, Maracanã, CEP: 20.550 – 013, Rio de Janeiro, RJ
Brasil

Resumo

*Neste trabalho apresentamos o algoritmo de ordenação Smoothsort [1][2], desenvolvido por Dijkstra em 1981 e que, inexplicavelmente, não é citado na literatura corrente sobre métodos de ordenação como um método relevante [3-5]. Sua importância vem do fato dele ter complexidade de pior caso O(*n log n*), restrita a O(*n*) para conjuntos de dados já ordenados, uma importante situação prática. Nenhum dos métodos destacados na literatura apresenta tal comportamento. Além de descrever o algoritmo, mostramos também análises alternativas sobre a complexidade do algoritmo, que não estão presentes no artigo original do método.*

Abstract

*We present the sorting algorithm Smoothsort [1] [2], developed by Dijkstra in 1981 which, inexplicably, is not mentioned in the current literature on methods of ordering as a special one[3-5]. Its importance comes from the fact that it has worst case complexity O(*n log n*) in general, restricted to O(*n*) for sets of data already ordered. This is an important practical situation. None of the methods described in the literature presents such behavior. In addition to describing the algorithm, we also show alternative analysis of algorithm complexity, which are not present in the original paper about the method.*

1. Introdução

Edsger Wybe Dijkstra deu contribuições importantes a áreas da computação tais como programação, sistemas operacionais, sistemas distribuídos, algoritmos em grafos. Ele criou, entre outros algoritmos, o famoso algoritmo que leva seu sobrenome, e é usado para achar o caminho mais curto de um nó aos demais nós em um grafo direcionado, cujas arestas tenham pesos positivos.

Grande parte de sua obra foi disseminada através de manuscritos e não pelos meios normais de divulgação científica (livros, revistas, anais de congressos). O algoritmo que será analisado neste trabalho, o *Smoothsort*, está documentado em [1] e [2]. Trata-se de um método avançado de ordenação, com desempenho equivalente ao dos melhores métodos de ordenação conhecidos, baseado no Heapsort [6], mas com a característica adicional de que ele é executado em tempo linear quando os dados já estão ordenados.

Inexplicavelmente, o Smoothsort não é apresentado na literatura sobre algoritmos de ordenação [3-5], nem mesmo considerando sua importância particular, dado a seu comportamento único. As poucas referências encontradas sobre o método [7,8] apresentam críticas ou na complexidade de sua implementação, ou no tempo de execução no caso médio, superior ao de outros métodos. Com isso, é deixada de lado uma idéia muito interessante para o aperfeiçoamento do algoritmo do Heapsort e uma estrutura de dados muito rica, as árvores de Leonardo, que se prestam a muitas análises combinatórias .

Neste artigo pretendemos resgatar um pouco dessa omissão incompreensível, apresentando e exemplificando o método. Além disso, mostramos análises relativas à complexidade do algoritmo. Um trabalho anterior de um dos autores, para a visualização do método pode ser visto em [9].

Na seção 2 apresentamos resumidamente o Heapsort, de onde derivou o Smoothsort. Apresentamos também as árvores de Leonardo, comparando-as aos Heaps. Na seção 3 descrevemos e exemplificamos o Smotthsort. Finalmente, na seção 4, apresentamos algumas análises sobre a complexidade do método.

2. Heapsort e árvores de Leonardo

O Heapsort, é baseado em uma estrutura de dados denominada heap. Um heap é uma árvore estritamente binária que satisfaz a seguinte propriedade: dado qualquer elemento da árvore, este será sempre maior ou igual que seus filhos diretos. Considerando a transitividade, cada elemento é maior ou igual que todos os seus descendentes na árvore. A árvore é, na verdade, implícita, simulada em um vetor, fazendo com que cada elemento no vetor com índice *i* seja pai dos elementos com índice *(i*2)+1* e *(i*2)+2* (considerando o endereço inicial do array igual a zero). A ordenação de um vetor com *n* elementos, pelo Heapsort é feita em dois passos:

a) transformação do vetor em um heap. Essa transformação visa atender à propriedade de heap, e é feita através de um procedimento denominado DesceHeap, ilustrado na Figura 1, para as letras do string 'ANDRE'. Ao final deste procedimento, a chave mais alta do vetor estará na posição inicial do mesmo.

b) execução de *n-1* passos, onde se troca o primeiro elemento do vetor com o último, e executa-se a função DesceHeap para o primeiro elemento do vetor. Esse processo é ilustrado na Figura 2.

O Heapsort é um excelente método de ordenação, tendo complexidade de pior caso igual a O(*n log n*), que é o melhor que se pode obter. Ele tem também complexidade de caso médio, O(*n log n*) e, por isso, é um método de ordenação de uso geral. Além disso, o método não exige memória adicional para a ordenação. A única melhoria possível seria no seu funcionamento na situação em que o vetor já está ordenado, ou praticamente ordenado. Quando o vetor está nessa situação, o Heapsort praticamente inverte os dados do vetor para, em seguida, inverter novamente e terminar o trabalho. Ou seja, o método não "percebe" que o vetor estava ordenado. Ocorre que a situação de ordenar um vetor já ordenado ou apenas com poucas inversões de dados é bastante comum na prática e normalmente deriva de acréscimo de alguns dados a um conjunto de dados já ordenado. A motivação para a criação do Smoothsort vem dessa observação. O que o criador do método procurou fazer foi inventar um algoritmo de ordenação que tivesse complexidade de pior caso igual a O(*n log n*) e complexidade O(*n*) quando o vetor já estiver ordenado.

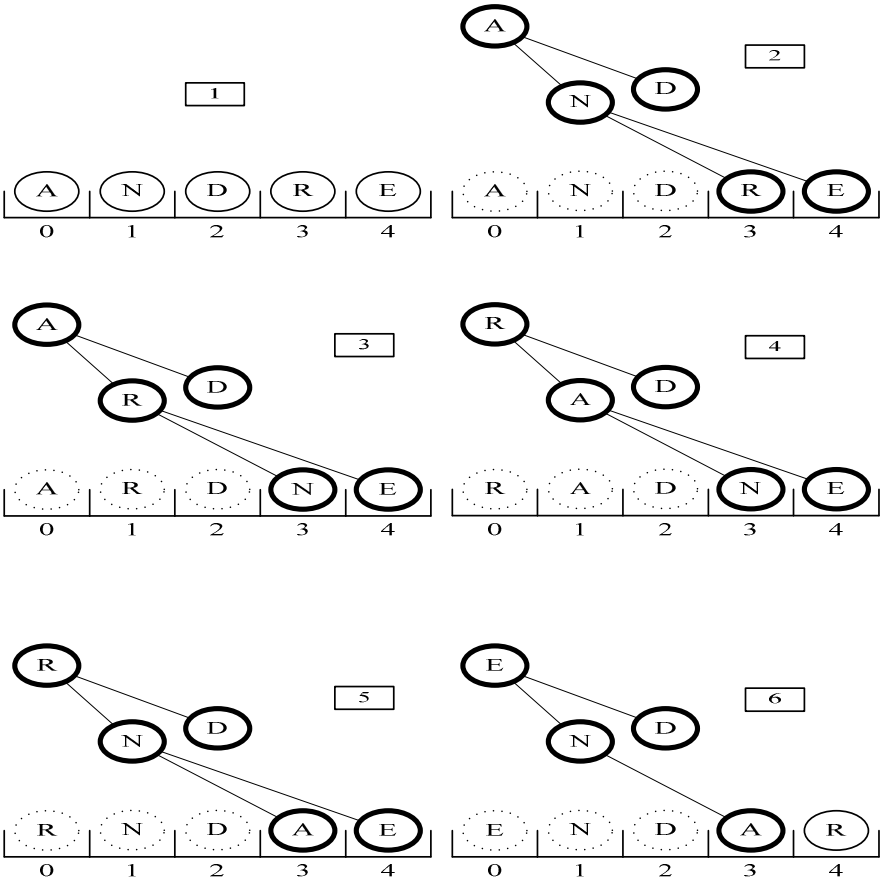

Figura 1. *Heapsort* em ação (1/2)

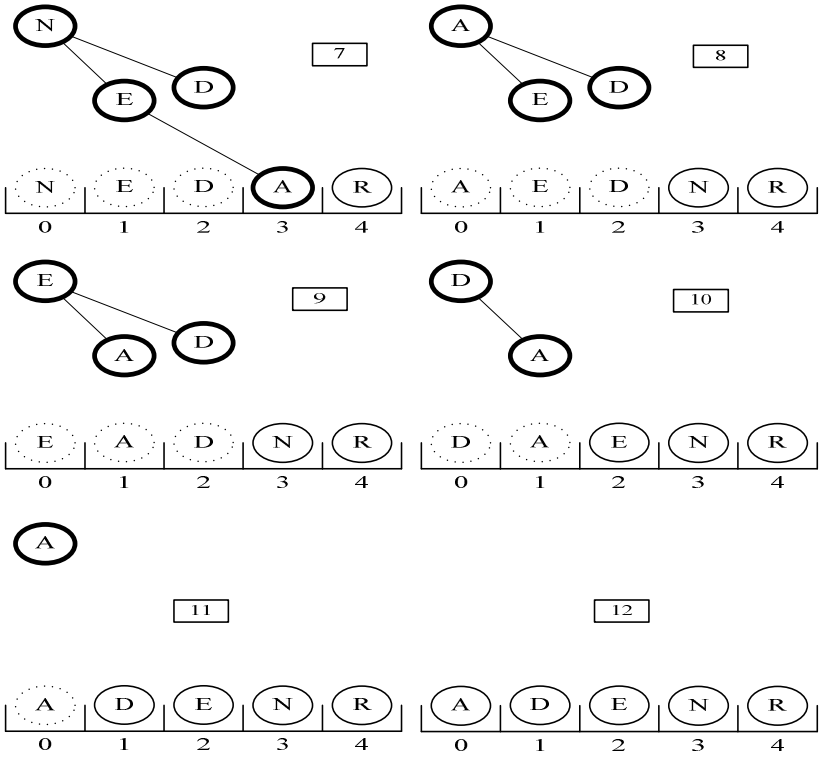

Figura 2. *Heapsort* em ação (2/2)

Vimos que a estrutura básica para o Heapsort é o heap, uma árvore binária "simulada" em um vetor. O "defeito" do Heapsort, no caso de vetores já ordenados pode ser explicado pelo fato da raiz da árvore estar no início do vetor. Nesse caso, para não ter que desordenar todo o vetor, o ideal é que a raiz da árvore (maior elemento) estivesse na direita do vetor. Essa foi a idéia que norteou a criação do Smoothsort. Não se conseguiu viabilizar a idéia apenas com uma árvore, mas com uma floresta, conforme descreveremos a seguir. Essa floresta é composta de árvores de Leonardo, árvores também implícitas, simuladas em um vetor. Uma árvore de Leonardo é uma árvore binária, onde o número de nós das subárvores são números consecutivos de Leonardo. Explicaremos os números de Leonardo a seguir.

Os números de Leonardo formam uma sequência parecida com a dos números de Fibonacci. Indicaremos por L_n o n-ésimo número de Leonardo e F_n o n-ésimo número de Fibonacci. Assim sendo, as fórmulas conhecidas para os números de Fibonacci podem ser usadas no estudo dos números de Leonardo sem muito esforço. Assim como os números de Fibonacci, os números de Leonardo também são definidos por uma recorrência:

$L_0 = 1, \quad L_1 = 1, \quad L_n = L_{n-1} + L_{n-2} + 1$

Os 20 primeiros números de Leonardo são os seguintes: 1, 1, 3, 5, 9, 15, 25, 41, 67, 109, 177, 287, 465, 753, 1219, 1973, 3193, 5167, 8361 e 13529. Pode-se mostrar que a relação entre L_n e F_n é dada por: $L_n = 2F_n - 1$.

Observando atentamente a recorrência que gera os números de Leonardo é fácil perceber que esses números podem ser usados para construir uma árvore binária de forma recursiva: $L_n = L_{n-1} + L_{n-2} + 1$

Cada número é formado somando-se dois números anteriores mais um, portanto é possível criar uma árvore binária onde a raiz da árvore tem L_{n-1} filhos à esquerda e L_{n-2} filhos à direita, conforme a Figura 3:

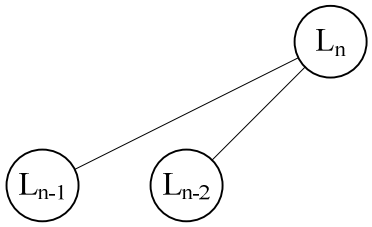

Figura 3. Árvore binária descrita pelos números de Leonardo

Com isso a árvore inteira tem L_n elementos, e como $L_n = L_{n-1} + L_{n-2} + 1$, tem-se que L_{n-1} corresponde à sub-árvore esquerda, L_{n-2} à sub-árvore direita e 1 corresponde à raiz da árvore, pois a raiz só tem um elemento.

A seguir, um exemplo numérico ilustra melhor como fazer uma árvore dessa forma. Na Figura 4 temos uma árvore binária com 9 elementos. Na raiz dessa árvore aparece o número nove, o maior valor. Os

antecessores de 9 na seqüência dos números de Leonardo são 5 e 3 e assim sucessivamente.

Esse tipo de representação em árvore pode ser feito de forma implícita, usando-se um array de elementos. A partir de agora, este tipo de árvore será chamada de árvore de Leonardo.

Ocorre que nem todos os arrays possuem dimensão igual a um dos números de Leonardo. Portanto, nem todos os arrays podem ser acessados como se fossem uma árvore de Leonardo implícita, mas podem ser acessados como se fossem um conjunto de árvores de Leonardo implícitas. Um conjunto de árvores de Leonardo é chamado de floresta de Leonardo. A Figura 5 ilustra uma floresta de Leonardo para um array com 8 elementos.

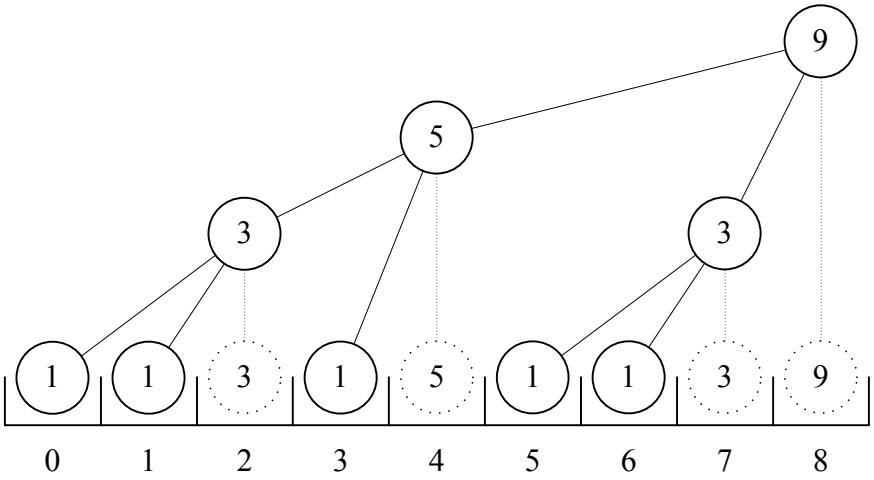

Figura 4. Árvore de Leonardo implícita sobre um array

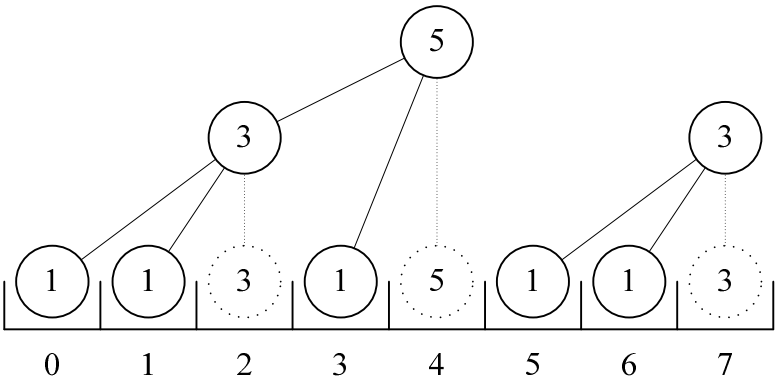

Figura 5. Floresta de Leonardo

A floresta de Leonardo que tenha o menor número de árvores pode ser criada de forma **g**ulosa. O procedimento guloso consiste em percorrer o vetor da esquerda para a direita e, para cada nova posição do vetor, considerar essa posição como a raiz, ou de uma árvore com apenas um nó, ou como uma nova raiz para as duas últimas árvores criadas, desde que os números de nós das duas representem números consecutivos (em ordem decrescente) de Leonardo. Desta forma, claramente a maior árvore de Leonardo para um vetor com *n* elementos corresponde à primeira árvore da floresta e o número de nós da mesma é o maior número de Leonardo contido em n. Sucessivamente isso vale para o restante do vetor.

No artigo original sobre o Smoothsort, Dijkstra deixou a prova desse problema para o leitor. Apresentamos a seguir uma prova simples, baseada no Teorema 1 e no Corolário 1 desse teorema, expostos em [10].

Teorema 1: (Cowen, Cowen & Steinberg)

"Suponha que $C_1 = \{a_1, a_2, ...a_k\}$ seja um conjunto de moedas guloso e seja $C_2 = \{a_1, a_2, ...a_k, a_{k+1}\}$. Seja $m = \lceil a_{k+1}/a_k \rceil$. Então C_2 é guloso sse $G(C_2, m.a_{k+1}) \leq m$."

O teorema acima relaciona-se com o problema clássico do Troco Mínimo, que consiste em, dados os tipos de moedas de um país e um valor x expresso em centavos, determinar o menor número de moedas para dar um troco de valor total x. Por exemplo, no Brasil, com o conjunto de moedas C = {1, 5, 10, 25, 50, 100}, o problema pode ser resolvido para qualquer x de forma gulosa. Se, digamos, o troco for de 36 centavos, escolhemos, primeiro uma moeda de 25, em seguida

uma de 10 e finalmente uma de 1. Neste caso diz-se que o conjunto de moedas C é guloso, e G(C,27)= 3, onde G indica o número mínimo de moedas necessárias. Os conjuntos de moedas nem sempre são gulosos, relativamente ao problema citado. Por exemplo, se C = {1, 9, 20, 50} e quisermos dar um troco de 36, a escolha gulosa levaria a 9 moedas (20+9+1+1+1+1+1+1+1), mas há uma solução mais simples com apenas 4 moedas (4 x 9). O corolário a seguir é autoexplicativo.

Corolário 1: (Cowen, Cowen & Steinberg)

"Qualquer conjunto de moedas formado pelos k primeiros números ímpares (1, 3, ...2k-1) é guloso."

O problema de determinar o número mínimo de árvores de Leonardo contidas em um valor *n* pode ser tratado exatamente como o problema do Troco Mínimo, onde as moedas são os números de Leonardo. O teorema a seguir garante que qualquer conjunto dos k primeiros números de Leonardo é guloso e, como consequência, fica provado o resultado desejado.

Teorema 2:

"O número de árvores de Leonardo criadas pelo algoritmo Smoothsort é mínimo."

Prova: Por indução. Inicialmente sabemos, pelo Corolário 1, que $C_3 = \{L_1, L_2, L_3\} = \{1, 3, 5\}$ é guloso. Suponhamos que, para dado $k > 3$, o conjunto dos k primeiros números de Leonardo $C_k = \{L_1, L_2, ...L_k\}$ seja guloso. Consideremos $C_{k+1} = \{L_1, L_2, ...L_k, L_{k+1}\}$. Temos $m = \lceil L_{k+1}/L_k \rceil = \lceil (L_{k-1}+L_k+1)/L_k \rceil = 2$, pois $L_{k-1}+1 < L_k$. Vamos mostrar que $G(C_{k+1},2L_k) = 2$. Realmente, o maior número de Leonardo contido em x $= 2L_k$ é L_{k+1}, pois $L_{k+2} = L_k+L_{k+1}+1 > 2L_k$. Portanto L_{k+1} é a "primeira moeda" a ser utilizada. Mas: $x-L_{k+1} = 2L_k-L_{k+1} = 2L_k-(L_k+L_{k-1}+1) = L_k-L_{k-1}-1 = L_{k-1}+L_{k-2}+1-L_{k-1}-1 = L_{k-2}$. Portanto, a segunda "moeda" a ser escolhida seria L_{k-2}, o que mostra que $G(C_{k+1},2L_k) = 2$. Pelo Teorema 1, C_{k+1} também é guloso. Isso finaliza a prova.

3. O Algoritmo *Smoothsort*

Nesta seção descreveremos resumidamente o funcionamento do Smoothsort. O algoritmo é parecido com o Heapsort, no sentido de que cria, em um primeiro passo, várias árvores, tal que as raízes das árvores devem ser maiores que todos os demais elementos, e a raiz de cada sub-árvore deve manter essa mesma propriedade. Para construir a floresta a partir do vetor, o Smoothsort considera um elemento do vetor de cada vez. Se as duas últimas árvores da floresta tiverem tamanhos correspondentes a dois números consecutivos da seqüência de Leonardo, então essas duas árvores junto com o elemento adicionado se juntam e formam uma nova árvore da floresta, conforme a recorrência: $L_n = L_{n-1} + L_{n-2} + 1$

Na floresta tem-se que L_{n-1} corresponde à penúltima árvore e L_{n-2} corresponde à última árvore. A Figura 6 ilustra bem esse processo de construção da Floresta de Leonardo para um vetor de 5 elementos.

A princípio, observando a construção dessa floresta com cinco elementos, pode-se pensar que as duas últimas árvores irão sempre se tornar filhas de um novo elemento que seja adicionado. No entanto, isto não é verdade. A Figura 7 ilustra esse fato mostrando uma floresta de 13 elementos com as árvores já formadas, mas com menos detalhes que a Figura 6.

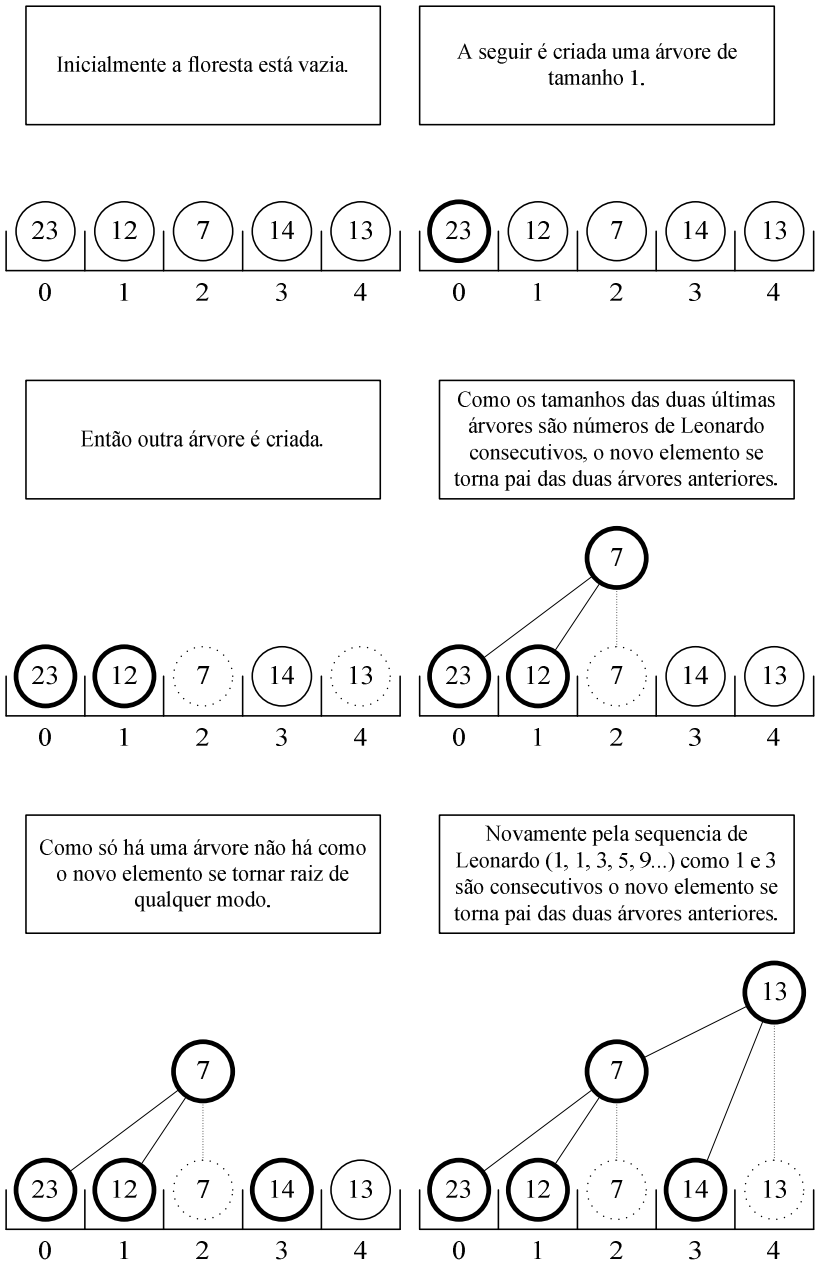

Figura 6. Formação da floresta

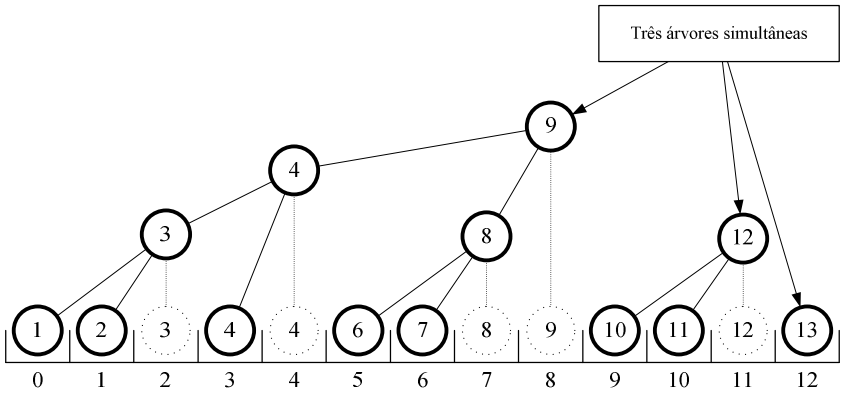

Figura 7. Três árvores na floresta

Seguindo-se esse processo simples é possível construir uma floresta de Leonardo implicitamente sobre arrays de qualquer tamanho. Para que os dados fiquem ordenados, o *Smoothsort* faz a floresta de árvores de Leonardo funcionar como se fosse um grande heap, de modo que a raiz de cada árvore de Leonardo sempre contém o maior elemento da árvore, e as raízes das sub-árvores também tenham essa mesma propriedade. No entanto como o Smoothsort não cria apenas uma árvore e sim várias árvores, é preciso que, além disso, haja alguma ordenação entre essas árvores. A ordenação que é feita consiste em fazer com que a raiz de uma árvore de Leonardo seja sempre maior ou igual às raízes de todas as outras árvores de Leonardo que estiverem à esquerda. A Figura 8 ilustra o processo, que utiliza uma função denominada DesceHeap.

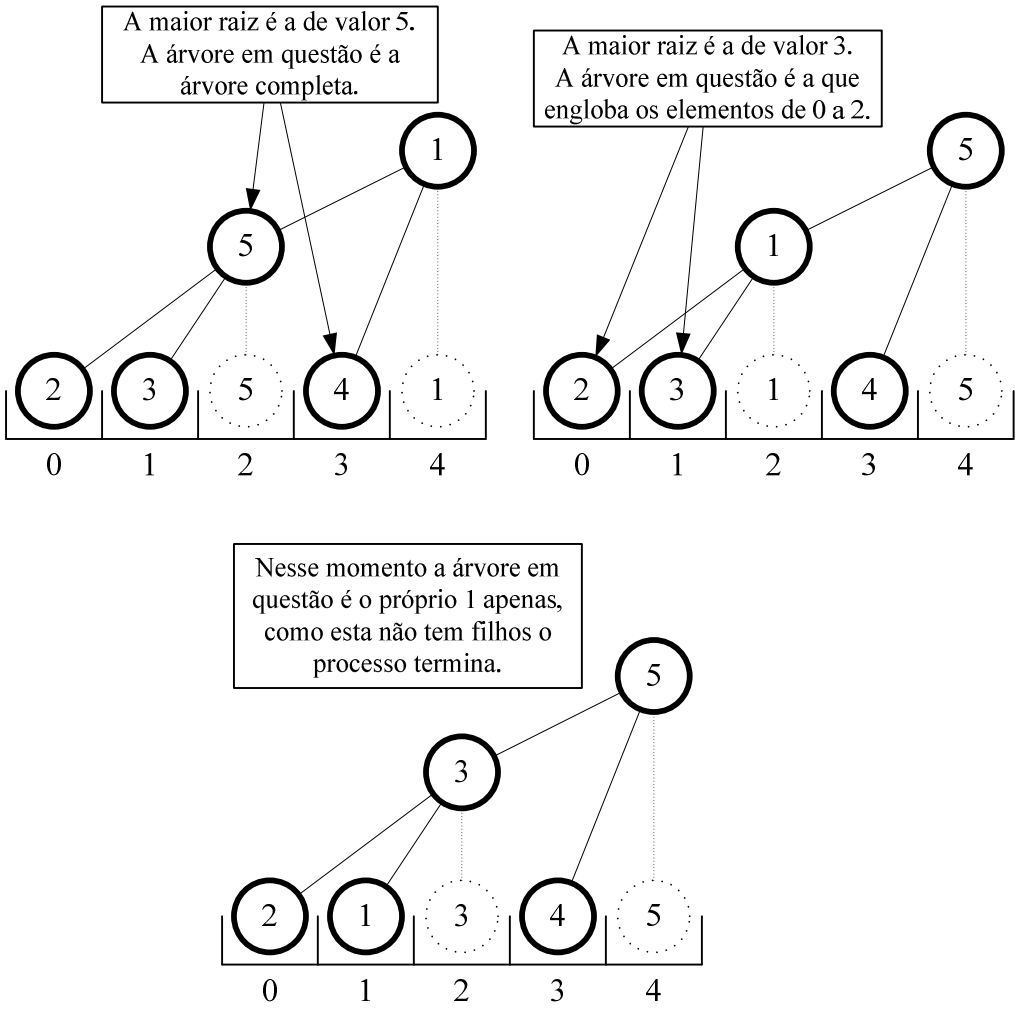

Figura 8. DesceHeap em ação

Na segundo passo faz-se a ordenação da floresta (função OrdenaFloresta), retirando-se sempre a raiz mais à direita do vetor e acertando as árvores restantes, de forma que a propriedade de heap continue valendo. Quando se retira a raiz mais à direita, ela é com certeza a maior chave do início do vetor até o ponto considerado. A sua retirada, quando a raiz é o único nó da árvore, diminui o número de árvores de Leonardo. Entretanto, se não for o nó único, as subárvores dessa árvore são acrescentadas ao conjunto de árvores de Leonardo. Neste caso, pode ser necessário acertar a ordenação das raízes, pois não se tem garantia que a introdução de duas novas árvores ao conjunto tenha essa propriedade necessária.

A Figura 10 ilustra o funcionamento dessa etapa do algoritmo.

4. Complexidade do Smoothsort

Conforme o Smoothsort vai construindo a Floresta de Leonardo, ele considera um elemento de cada vez do vetor e o insere na floresta. Ao desfazer a Floresta, o Smoothsort retira um elemento de cada vez da Floresta. Portanto, a complexidade do Smoothsort é composta por uma parte linear, relativa à colocação e remoção dos elementos na Floresta somada a outras duas partes, a saber:

1) O número de elementos do vetor vezes o custo de inserção na Floresta.

2) O número de elementos do vetor vezes o custo de remoção da Floresta.

Para inserir ou remover um elemento na Floresta, o Smoothsort pode executar a função DesceHeap e também a função OrdenaFloresta. Então, para sabermos a complexidade do Smoothsort temos que saber a complexidade dessas funções.

Para sabermos a complexidade da função DesceHeap, basta sabermos qual é a altura de uma árvore qualquer de Leonardo. Para tanto, isto é, para acharmos a profundidade máxima de uma árvore de Leonardo, devemos procurar pelo caminho que desce pelo filho da esquerda, pois este caminho é que nos dirá qual é a altura da árvore. Podemos facilmente perceber que esse valor corresponde ao índice do número de Leonardo correspondente à árvore. Exemplificando: se uma árvore tem tamanho 5, sua altura será 3, pois 5 é o terceiro número de Leonardo. A fórmula fechada vem da relação $L_n = 2F_n - 1$. Temos, para Fibonacci:

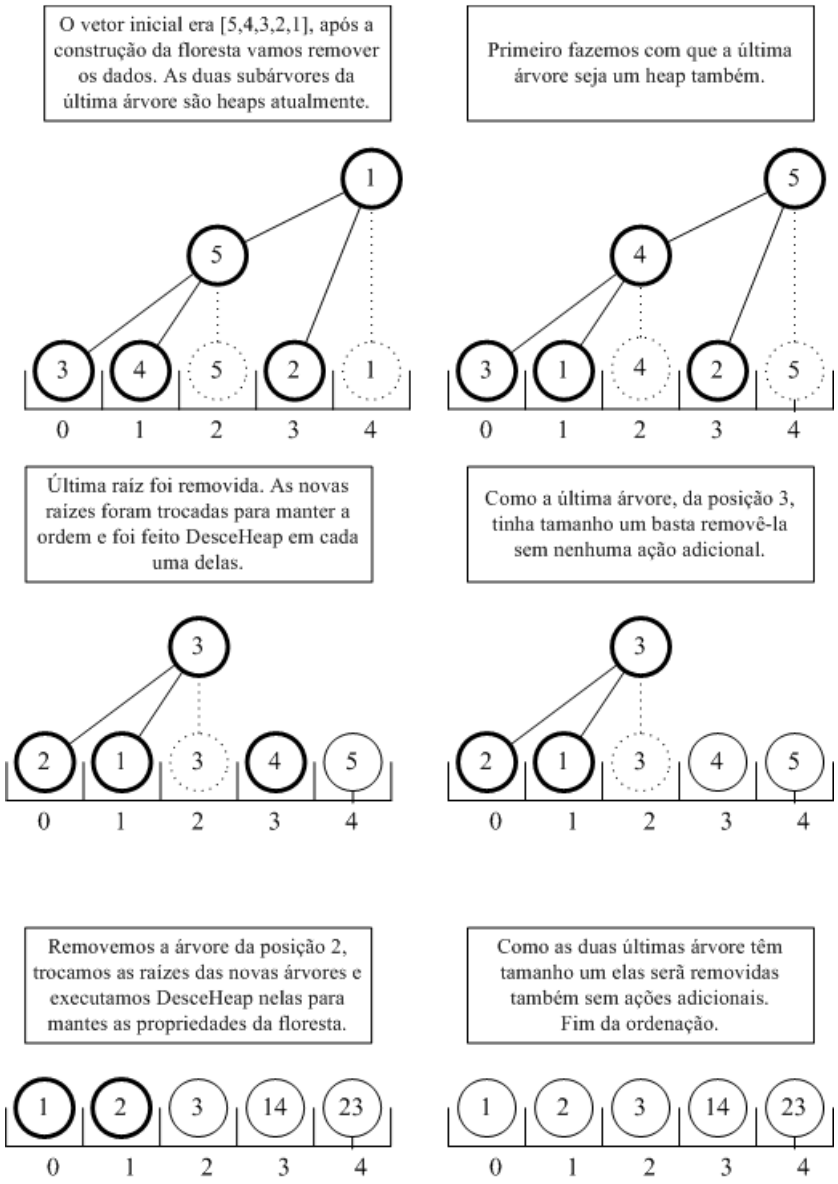

Figura 9. Ilustração da Função Ordena Floresta

$F(n) = \frac{\alpha^n - (1-\alpha)^n}{\sqrt{5}}$, sendo α a razão áurea, cujo valor é de aproximadamente 1.618034. Portanto, $L(n) = 2\frac{\alpha^n - (1-\alpha)^n}{\sqrt{5}} - 1$. Basta encontrar a inversa desta fórmula para acharmos a complexidade do DesceHeap. Como $\lim_{n\to\infty}(1-\alpha)^n = 0$, já que $1-\alpha \cong$ -0.61803, podemos ignorá-lo para simplificar nossos cálculos. Ficamos com:

$$L^{-1}(L(n)) \cong \log_\alpha\left(\frac{n\sqrt{5}}{2}\right) - 1 \cong \log_\alpha n + \log_\alpha\left(\frac{\sqrt{5}}{2}\right) - 1$$

Cabe ressaltar que mesmo essa fórmula não sendo exata, pois desprezamos o termo que tende a zero, ela converge muito rapidamente. Por exemplo, usando o Excel para calcular $L^{-1}(392835)$ obtemos o valor 25.99999 que é muito próximo de 26, e calculando $L^{-1}(635621)$ já obtemos como resultado o valor 27 que é o valor correto. Daí em diante os resultados ficam todos exatos. A partir desta expressão, podemos achar o valor de n, que é igual à altura da árvore, dado o tamanho da árvore. Mas como o nosso objetivo é tão somente determinar a complexidade do algoritmo, podemos descartar todas as constantes da nossa fórmula, ficando com a complexidade $O(\log_\alpha n)$

Agora vamos descobrir a complexidade da função OrdenaFloresta. Como sabemos, o que a função OrdenaFloresta faz, é ir "movendo" a raiz de uma árvore para a esquerda, enquanto a árvore da esquerda tiver uma raiz maior do que a raiz sendo movida. Sendo que no final do processo, esta função executa a função DesceHeap na árvore onde essa raiz tiver ido parar.

Portanto, a complexidade desta função é uma soma de duas partes. A primeira consiste, na complexidade da operação de mover a raiz da árvore em questão, de árvore em árvore para a esquerda. Já a segunda consiste em executar a função DesceHeap no fim do processo. Como já sabemos a complexidade da segunda parte vamos analisar apenas a primeira.

Como sabemos, o vetor será dividido em várias árvores de Leonardo, de forma gulosa. Assim, a árvore mais à esquerda será a maior de todas, pois o tamanho da mesma, será o maior número de Leonardo que seja menor ou igual ao total de elementos do vetor. A árvore seguinte, caso tenha sobrado algum elemento, terá um tamanho igual ao maior número de Leonardo que seja menor ou igual ao total de elementos restantes. E assim sucessivamente, até não sobrar nenhum elemento.

Queremos saber quantas árvores têm na Floresta para um vetor de tamanho n. Cabe ressaltar que o número de árvores varia conforme a Floresta vai sendo construída e destruída, então o ideal é sabermos qual o maior número de árvores que existiu na Floresta simultaneamente, em qualquer instante durante o processo de ordenação.

Um fato que sabemos, é que nunca existirão mais de duas árvores de tamanhos consecutivos em qualquer instante. Isto acontece por uma combinação de fatores. O primeiro fator é que as árvores têm tamanhos decrescentes, da esquerda para a direita, pois seus tamanhos equivalem a uma escolha gulosa de tamanhos, que englobe o maior número possível de elementos sendo considerados num dado instante. O segundo fator é que estando as árvores em ordem decrescente da esquerda para a direita, a única forma de existirem três árvores de tamanhos consecutivos seria com essas árvores estando todas juntas. Mas isso jamais irá acontecer devido ao funcionamento do algoritmo: ao inserir um novo elemento na Floresta, é verificado se as duas últimas árvores da Floresta têm tamanhos que sejam números consecutivos de Leonardo, e caso sejam, essas duas últimas árvores irão se fundir, e o novo elemento vai ser a raiz de uma nova árvore com essas duas sendo suas filhas.

Assim, temos um teto máximo para o número de árvores da Floresta. Esse teto corresponde ao tamanho de uma seqüência decrescente de números de Leonardo, onde o primeiro número é menor ou igual ao total de elementos do vetor, e não pode haver três números de Leonardo consecutivos na seqüência.

Com isso podemos estabelecer um teto um pouco mais alto, porém mais fácil de trabalhar. Podemos ignorar a regra de que não podem existir três árvores consecutivas, cujos tamanhos sejam números consecutivos de Leonardo.

E então, temos que a maior seqüência possível de árvores, tem tamanho igual a uma seqüência de números de Leonardo decrescente, que começa pelo primeiro número de Leonardo maior ou igual que o tamanho do vetor. Nesse caso, é maior ou igual porque queremos um teto, e essa seqüência decresce até o primeiro número de Leonardo.

Com essa simplificação, temos que a Complexidade da função OrdenaFloresta será portanto igual à complexidade da função DesceHeap, somada à complexidade da própria função DesceHeap, já que a função OrdenaFloresta executa a função DesceHeap no final. Porém, como somamos duas complexidades iguais, podemos colocar o fator 2 em evidência como uma constante e cortá-lo.

Assim, finalmente temos que a complexidade da função OrdenaFloresta é igual a complexidade da função DesceHeap. Como cada uma dessas operações pode ser executada para cada um dos elementos do vetor, chegamos finalmente à complexidade do algoritmo de ordenação Smoothsort que é igual a: $O(n \log_\alpha n)$

É importante ressaltar que esta é a complexidade do pior caso, pois no melhor caso, quando os dados estão todos ordenados a complexidade é *O(n)*.

Neste caso, ao se executar a operação DesceHeap os elementos não irão descer nem um nível sequer no heap. Pois as árvores formadas pelo Smoothsort têm raiz para a direita e os elementos para a esquerda, assim se os dados já estiverem ordenados a raiz da árvore já será o maior elemento de todos. Portanto, todas as árvores implícitas formadas pelo Smoothsort já serão heaps, e na desconstrução da Floresta o mesmo irá ocorrer.

Já com relação à Função OrdenaFloresta algo similar irá acontecer se os dados já estiverem ordenados. A raiz da árvore por onde a função OrdenaFloresta irá começar a executar já será maior que todas as raízes das árvores anteriores, não necessitando assim de nenhuma troca de elementos. E depois quando esta função executar a função DesceHeap essa função será executada em tempo constante conforme foi explicado.

Com relação à complexidade média, no caso dos dados estarem semi-ordenados, o que irá ocorrer é que algumas execuções da operação DesceHeap e algumas execuções da operação OrdenaFloresta irão executar em tempo constante, enquanto outras irão executar em tempo $O(n \log_\alpha(n))$, fazendo assim com que a complexidade do Smoothsort varie de forma suave, indo de uma execução em tempo linear quando os dados estão ordenados, até uma execução em tempo $O(n \log_\alpha n)$ no pior caso, passando por todos os pontos intermediários entre essas duas complexidades.

A complexidade de espaço do Smoothsort pode agora ser inferida a partir da complexidade da função OrdenaFloresta. Além do array para os *n* elementos a serem ordenados, o algoritmo precisa de guardar dados auxiliares. Como vimos, a função OrdenaFloresta transporta a raiz da árvore da direita para a esquerda em $\log_\alpha n$ passos. É usado um número de Leonardo para guardar o tamanho de cada árvore da Floresta de Leonardo. Então teremos que armazenar tantos números de Leonardo quantas árvores possam ter na Floresta. Com essas duas informações chegamos à conclusão que o memória adicional requerida, descrita na notação de ordem de grandeza é $O(\log_\alpha n)$. Portanto, a memória adicional não muda a complexidade de espaço devida ao vetor, que continua sendo *O(n)*.

5. Conclusões

Embora o Smoothsort tenha sido inventado há bastante tempo, o método é um aperfeiçoamento interessante do Heapsort, tendo complexidade de pior caso igual ao dos melhores algoritmos de ordenação (Heapsort, Mergesort), complexidade de caso médio também igual à desses melhores métodos (incluindo aqui também o Quicksort), ele é o único dos "grandes" métodos que apresenta complexidade linear quando os dados já estão ordenados, uma importante situação prática.

Neste artigo procuramos resgatar um pouco desse esquecimento, ilustrando o funcionamento do método e apresentando a rica estrutura de dados usada, as árvores de Leonardo. Contribuimos também com uma prova original que justifica o procedimento guloso para criar as árvores e com análises alternativas sobre a complexidade do algoritmo.

Referências

[1] Dijkstra, E. W. (1981) “Smoothsort, an alternative for sorting in situ”, em http://www.cs.utexas.edu/users/EWD/ewd07xx/EWD796.PDF, visitado em outubro de 2009.

[2] Dijkstra, E. W. (1981) “Smoothsort, an alternative for sorting in situ”, em http://www.cs.utexas.edu/users/EWD/ewd07xx/EWD796a.PDF, visitado em outubro de 2009.

[3] Cormen, T. H.; Leiserson, C. E.; Rivest R. L.; Stein, C. Introduction to Algorithms. 3rd edition, McGraw-Hill, New York, (2009).

[4] Sedgewick, R. Algorithms in C, Parts I-IV. 3rd edition, Addison Wesley, New York, (2001).

[5] Skiena, S. The Algorithm Design Manual. 2nd edition, Springer-Verlag, New York, (2008).

[6] Williams, J.W.J. "**Algorithm 232 Heapsort"**, CACM, 7(6):347-348, Jun 1964.

[7] Bron, C; Hesselink, W.H. "Smoothsort Revisited" in *Information Processing Letters* (39), pp- 269-276, (1991).

[8] Cunningham & Cunningham Inc; "Smooth Sort", em http://c2.com/cgi/wiki?SmoothSort, visitado em dezembro 2009.

[9] Pfitzner, A.L; "Visualizador do Smootsort", em http://www.ime.uerj.br/~pauloedp/ESTD/EDPagina.html, visitado em dezembro 2009.

[10] Cowen, L.J.; Cowen, R.; Steinberg A. "Totally Greedy Coin Sets and Greedy Obstructions". The electronic Journal of Combinatorics 15, (2008).