Construção e Análise de Algoritmos

aula 04: O algoritmo de ordenação Heapsort

1 Introdução

Vejamos o que nós fizemos até aqui.

Primeiro, nós consideramos um algoritmo de ordenação ingênuo (o *algoritmo da bolha*) que, apesar de fazer o seu trabalho corretamente, não tem um desempenho muito bom: ele executa em tempo $O(n^2)$, o que corresponde essencialmente a comparar todos os pares de elementos da lista.

Tomando esse algoritmo como base, nós aplicamos pela primeira vez a *técnica da divisão*: nós dividimos a lista entre as posições pares e ímpares e executamos o algoritmo da bolha nas duas partes. Essa ideia, por si só, não é suficiente para reduzir o tempo de execução do algoritmo. Mas, seguindo por esse caminho, nós acabamos chegando ao algoritmo *Shellsort*, que executa em tempo $O(n \log^2 n)$ — uma melhoria e tanto!

O próximo passo foi aplicar a estratégia de divisão natural: quebrar a lista no meio e ordenar as duas metades em separado. Essa etapa precisa ser complementada pelo procedimento de *intercalação*, que combina as duas metades para produzir uma lista completamente ordenada. Essa estratégia reduz o tempo de execução do algoritmo aproximadamente à metade. E, dado que ela funciona, a estratégia de divisão foi aplicada recursivamente para obter o algoritmo *Mergesort*, que executa em tempo $O(n \log n)$.

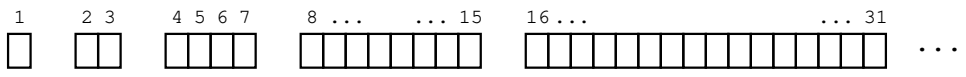
Apesar do seu desempenho excelente, o algoritmo Mergesort ainda tem um problema: em sua implementação típica, o procedimento de intercalação requer uma quantidade de memória auxiliar comparável ao tamanho da lista que está sendo ordenada. A solução para esse problema veio na forma de uma pequena modificação na estratégia de divisão: ao invés de quebrar a lista no meio, nós aplicamos o procedimento de *partição*, que move os menores para a esquerda e os maiores para a direita. Esse procedimento, em geral, não produz uma divisão perfeitamente balanceada da lista. Mas, a sua aplicação recursiva é suficiente para garantir que o algoritmo *Quicksort* execute em tempo médio $O(n \log n)$.

Nesse ponto, nós deveríamos estar satisfeitos. Nós conseguimos um algoritmo que não utiliza memória auxiliar e que, na prática, executa tão rápido quanto qualquer outro. Mas, ainda fica a dúvida se não existiria um algoritmo que não utiliza memória auxiliar e executa em tempo $O(n \log n)$ mesmo no pior caso.

Para resolver essa questão, nós vamos examinar a seguir mais uma estratégia de divisão.

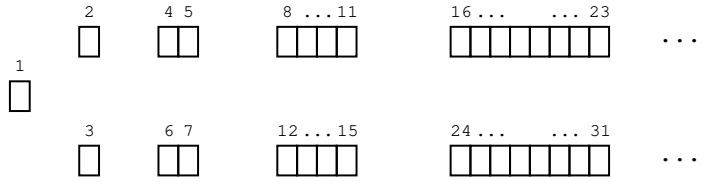
2 O algoritmo de ordenação Heapsort

Imagine dessa vez que a lista é dividida em blocos da seguinte maneira

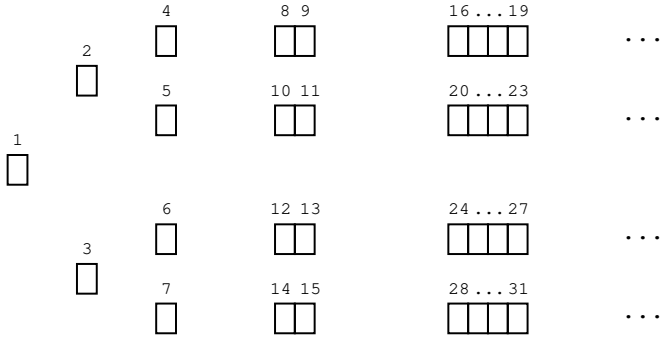


Isto é, cada bloco tem o dobro do tamanho do anterior, e nós temos um total de $\log_2 n$ blocos.

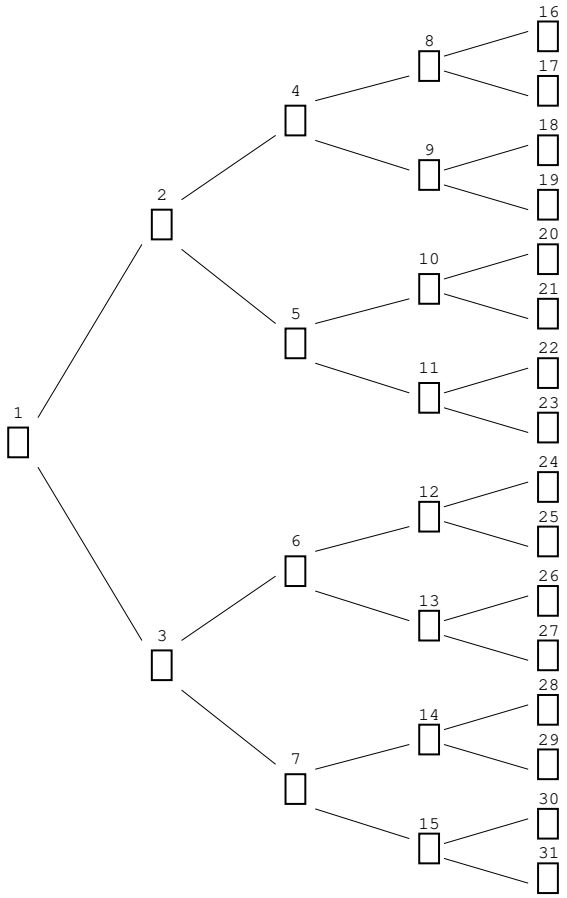
A seguir, cada bloco de tamanho maior do que 1 é dividido novamente na metade



Esse procedimento é aplicado mais uma vez



E continuamos fazendo isso até que todos os blocos tenham tamanho 1.



Apesar da aparência complicada, esse esquema é completamente regular:

- percorrendo as colunas da esquerda para a direita, nós temos a sequência 1, 2, 3, ..., 31
- e cada posição k tem à sua esquerda a posição $k/2$ e à sua direita as posições $2k$ e $2k+1$

Essa estratégia divide a lista em uma coleção de sublistas:

- `1, 2, 4, 8, 16`
- `1, 2, 4, 8, 17`
- `1, 2, 4, 9, 18`
- ...
- `1, 3, 7, 15, 31`

E a novidade aqui é que as sublistas possuem elementos repetidos.

Por exemplo, as duas primeiras sublistas são iguais, com a exceção do último elemento.

O próximo passo consiste em ordenar todas as sublistas.

Como existem $n/2$ sublistas e todas elas tem tamanho $\log_2 n$, a execução do algoritmo da bolha sobre cada uma delas leva tempo total $n \log^2 n$.

Isso não é tão mal assim.

Mas, fazendo isso, nós já excedemos o tempo dos algoritmos que vimos nas aulas anteriores.

De fato isso não é necessário, e é aqui que as coisas começam a ficar interessantes ...

Como as duas primeiras sublistas são praticamente iguais, ao ordenar a primeira a segunda fica quase que completamente ordenada também.

E a terceira, que também é muito parecida com as duas primeiras, fica bastante próxima de estar ordenada.

Essa observação vai nos permitir ordenar todas as sublistas de maneira eficiente.

Vejamos.

Após ordenar a primeira sublista, apenas o elemento da posição `17` está fora de ordem (possivelmente) na segunda sublista.

Mas, para colocá-lo em ordem, basta fazer uma varredura (da direita para a esquerda) a partir desse elemento

```
1   2   4   8   17
            ⟵
```

(Observe que a ordenação da segunda sublista não desordena a primeira — porque?)

Da mesma forma, para colocar a terceira sublista em ordem, basta fazer primeiro uma varredura a partir do elemento da posição `9`, e depois uma varredura a partir do elemento da posição `18`.

```
1   2   4   9   18
        ⟵
            ⟵
```

Agora, note que, enquanto a ordenação da primeira sublista (pelo algoritmo da bolha) levou tempo $\log^2 n$, a segunda sublista foi ordenada em tempo $\log n$, e a terceira em tempo $2 \log n$.

A questão agora é saber qual o tempo total necessário para ordenar todas as sublistas por esse método.

E a resposta é simples!

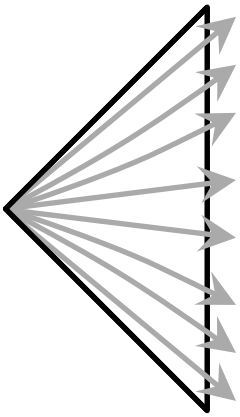
Note que cada elemento fora de ordem é colocado no seu lugar correto por meio de uma única varredura.

Cada varredura leva tempo $\log n$, e não existem mais do que n elementos fora de ordem.

Logo, fazendo as coisas na ordem correta, todas as listas são ordenadas em tempo $O(n \log n)$.

Agora sim!

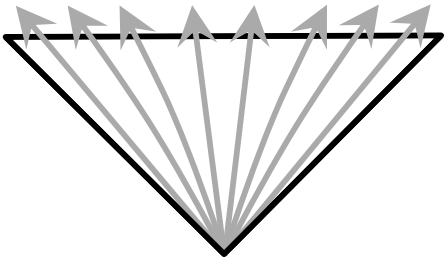
Após a ordenação de todas as sublistas, a nossa estrutura fica mais ou menos assim



E a nossa tarefa, a seguir, é ordenar completamente a lista a partir desse ponto.

Para usar uma analogia física, nós vamos fazer isso por *gravidade*.

O primeiro passo consiste em colocar a estrutura nessa posição:



Em seguida, nós vamos fazer os elementos *escoarem* pela parte de baixo.

Isto é, nós já sabemos que o menor elemento de todos se encontra na posição `1` (i.e., no vértice de baixo).

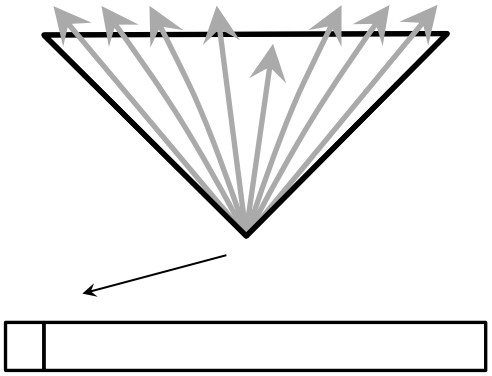
A ideia, então, é movê-lo para a primeira posição de uma lista auxiliar, deixando um buraco no seu lugar.

Esse buraco será preenchido (por gravidade) por um dos elementos que se encontram imediatamente acima dele; mais especificamente, pelo menor deles.

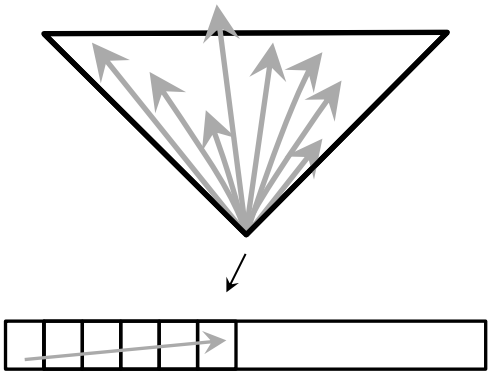
Quando esse elemento é movido para a primeira posição, ele próprio deixa um buraco em seu lugar.

E esse buraco é mais uma vez preenchido por gravidade.

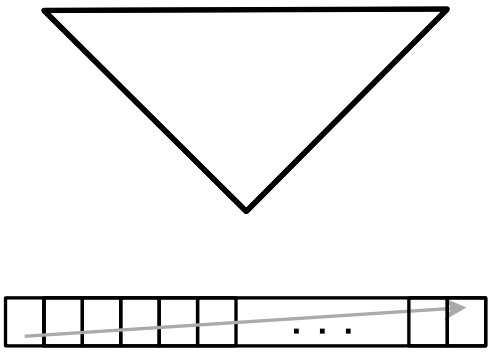
O efeito final é que uma das sublistas é deslocada uma posição para baixo.



E a ideia agora é repetir essa operação várias vezes



Até que no final a estrutura esteja vazia e nós tenhamos uma lista completamente ordenada.



Quanto tempo leva isso?

Bom, quando removemos o elemento da posição `1`, o buraca que ele deixa é preenchido por um dos dois elementos que ficam acima dele.

E é preciso uma comparação para verificar qual é o menor deles.

O buraco deixado pelo menor elemento também requer uma comparação para ser preenchido.

E a coisa continua assim até que alcançamos o topo da estrutura.

Como a estrutura tem altura $\log n$, o tempo total dessa operação é $logn$.

Finalmente, como são n elementos a serem escoados, a coisa toda leva tempo $n \log n$.

Isso é ótimo!

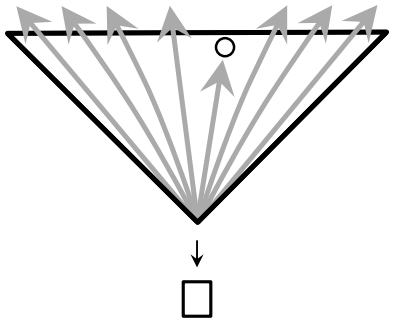
Como a etapa de construção da estrutura leva tempo $n \log n$, e a etapa de escoamento da estrutura também leva tempo $n \log n$, nós temos um algoritmo de ordenação que executa em tempo

$$n \log n \ + \ n \log n \quad = \quad 2n \log n \quad = \quad O(n \log n)$$

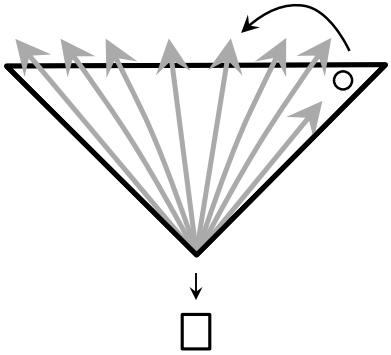
Sim, isso é muito bom, mas nós ainda não resolvemos o problema da memória auxiliar.

Agora é que nós vamos poder apreciar a beleza da estrutura que foi utilizada ...

Voltando alguns passos atrás e examinando a remoção do primeiro elemento da estrutura, nós observamos que ela deixa um buraco na parte de cima:

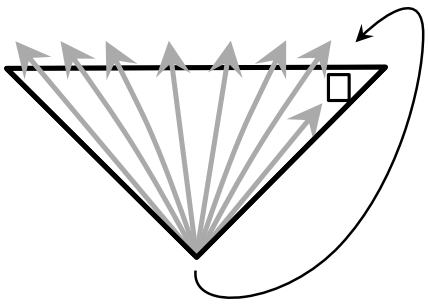


A ideia é que o último elemento da lista (que fica no vértice direito da estrutura) pode ser movido para essa posição.

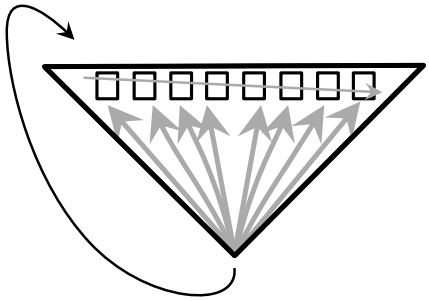


(Ao fazer isso, será preciso uma nova varredura, pois o elemento movido pode estar fora de ordem na nova sublista.)

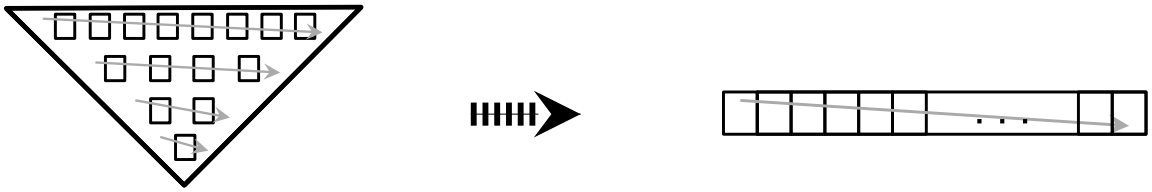
Finalmente, o elmento que está sendo removido da estrutura pode ser colocado nessa nova posição.



Repetindo essa operação várias vezes, os elementos removidos vão formando uma lista ordenada no topo da estrutura:



Até que, no final, a estrutura inteira corresponde a uma única lista ordenada:



Um detalhe curioso é que no final a lista foi ordenada de maneira decrescente.

Esse pequeno problema pode ser resolvido de duas maneiras:

- inverter a ordem dos elementos na lista (em tempo $O(n)$), ou
- ordenar as sublistas de maneira decrescente na etapa de construção da estrutura

Em qualquer caso, nós obtemos um algoritmo de ordenação que executa em tempo $O(n \log n)$ e não utiliza memória auxiliar.

3 Implementação do algoritmo Heapsort

Como todos já sabem, a estrutura que foi utilizada para ordenar a lista na seção anterior é chamada *Heap*.

Na sua versão MAX, o heap é uma organização dos elementos de um vetor `V[1..n]` que satisfaz a seguinte propriedade:

- para cada posição k, o elemento `V[k]` é maior ou igual aos elementos `V[2k]` e `V[2k + 1]`

Essa estrutura de dados suporta as operações de inserção de um novo elemento e remoção do maior elemento de maneira extremamente eficiente.

Nós chegamos ao algoritmo Heapsort por meio de uma estratégia nova de divisão da lista, mas ele também pode ser descrito da seguinte maneira:

- primeiro, a lista desordenada `V[1..n]` é transformada em um heapMax
- depois, nós removemos sucessivamente o maior elemento do heapMax, e o colocamos na posição que fica vazia após a remoção

A construção do heapMax pode ser feita também por meio de inserções sucessivas.

Abaixo nós temos o pseudo-código do procedimento de inserção:

```
Procedimento Inserção-HeapMax ( x, V[1..n], m )
 {
   Se ( m = n )   Retorna     // o heap já está cheio

   m ← m + 1
   V[m] ← x                  // insere o elemento na última posição

   k ← m
   Enquanto ( k > 1 )        // propaga elemento para cima, até posição correta
    {
      pai ← k/2

      Se ( V[pai] > V[k] )   Retorna

      Troca (pai,k)
      k ← pai
    }
 }
```

E abaixo nós temos o procedimento que constrói o heapMax:

```
Procedimento Heapfy ( V[1..n] )
 {
   Para i ← 2 Até n

       Inserção-HeapMax ( V[i], V[1..n], i-1 )
 }
```

Examinando o código do procedimento de inserção, nós observamos que o seu laço pode executar no máximo $\log_2 n$ iterações pois, o valor inicial da variável `k` é no máximo n, e a cada iteração o seu valor é reduzido à metade.

Portanto, o procedimento `Inserção-HeapMax` executa em tempo $O(\log n)$.

A seguir, nós observamos que o procedimento `Heapfy` realiza $n-1$ chamadas ao procedimento de inserção.

Logo, a construção do heapMax leva tempo $O(n \log n)$.

A próxima etapa da ordenação é baseada no procedimento de remoção.

Abaixo nós temos o pseudo-código desse procedimento.

```
Procedimento Remoção-HeapMax ( V[1..n], m )
 {
   Se ( m = 0 )   Retorna     // o heap está vazio

   aux ← V[1]                 // armazena o maior elemento
```

```
    V[1] ← V[m]                // e coloca o último elemento na sua posição
    m ← m - 1

    k ← 1
    Enquanto ( 2k ≤ m )        // propaga elemento p/ baixo, até posição correta
     {
       filho1 ← 2k;   filho2 ← 2k + 1

       Se ( m = 2k   ou   V[filho1] > V[filho2] )
        {
          Se ( V[k] > V[filho1] )   Break

          Troca (k,filho1)
          k ← filho1
        }
       Senão
          Se ( V[k] > V[filho2] )   Break

          Troca (k,filho2)
          k ← filho2
        }
     }
    Retorna aux
  }
```

Um argumento semelhante ao que foi apresentado acima mostra que o tempo de execução do procedimento de remoção também é $O(\log n)$.

Finalmente, abaixo nós temos o pseudo-código do algoritmo Heapsort

```
Procedimento Heapsort ( V[1..n] )
 {
   Heapfy ( V[1..n] )

   Para i ← n até 2
    {
      aux ← Remoção-HeapMax ( V[1..n], i )

      V[i] ← aux
    }
 }
```

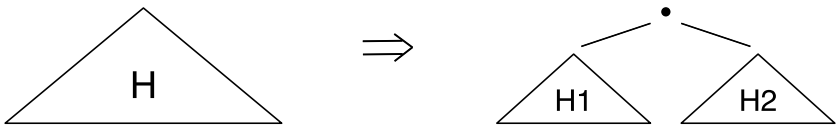
Como ele realiza uma chamada ao procedimento `Heapfy` e $n-1$ chamadas ao procedimento de remoção, seu tempo de execução é

$$O(n \log n) \;+\; (n-1) \cdot \log_2 n \quad = \quad O(n \log n)$$

Exercícios

1. **Heapfy recursivo**

   Note que um heap pode ser decomposto em dois heaps menores mais um elemento



   onde o elemento isolado é `V[1]`, o heap `H1` consiste no elemento `V[2]` e todos os seus desecendentes, e o heap `H2` consiste no elemento `V[3]` e todos os seus desecendentes (veja a figura na página 2).

   Essa observação nos dá uma estratégia alternativa para construir o heap:

   - construa o heap `H1`
   - construa o heap `H2`
   - coloque o elemento em `V[1]` na sua posição correta (propagando ele para baixo)

   A ideia, é claro, é aplicar essa estratégia recursivamente na construção dos heaps `H1` e `H2`.

   a) Apresente o pseudo-código de um algoritmo recursivo que constrói um heapMax como indicado acima.

   b) Analise o tempo de execução do seu algoritmo

2. **Mergesort inplace** (1 ponto extra na AP1[1])

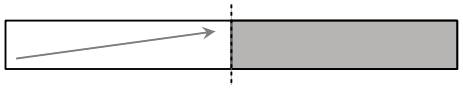
   Nós vimos na aula 02 que o algoritmo Mergesort requer o uso de uma memória auxiliar do mesmo tamanho da lista que está sendo ordenada.

   Mas, isso só vale para a implementação padrão do procedimento de intercalação.

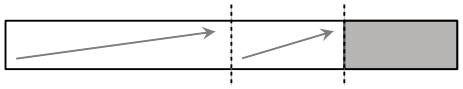
   Fazendo as coisas de maneira inteligente, é possível eliminar o uso da memória auxiliar.

   Vejamos.

   A primeira observação é que a primeira metade da lista pode ser ordenada (via mergesort) utilizando a segunda metade (ainda desordenada) como memória auxiliar.



   De maneira análoga, o terceiro quarto da lista pode ser ordenado (via mergesort) utilizando o último quarto como memória auxiliar.



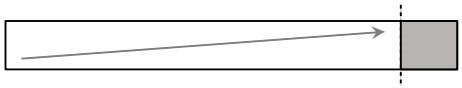
[1] para quem apresentar a solução na próxima aula

E a ideia agora é fazer a intercalação das duas partes ordenadas A e B, utilizando a parte desordenada C como memória auxiliar.

Se você conseguir fazer isso, então o problema está basicamente resolvido.

Isto é, em seguida nós ordenamos o sétimos oitavo da lista (via mergesort) utilizando o último oitavo como memória auxiliar

e depois repetimos o novo procedimento de intercalação



para aumentar a porção ordenada da lista.

Continuando dessa maneira, a lista inteira ficará completamente ordenada.

a) Descreva em detalhe o funcionamento do procedimento de intercalação mencionado acima.

b) Analise o tempo de execução do algoritmo Mergesort quando esse novo procedimento de intercalação é utilizado.