TimSort

Aluno: Caio de Miranda
Orientador: Profº Renato Carmo

Universidade Federal do Paraná (UFPR)
Departamento de Informática (DInf)
Grupo de Pesquisa em Teoria da Computação, Otimização e Combinatória (TEORIA)

10 de dezembro de 2025

Sumário

Sumário

Introdução

- Criado por Tim Peters em 2002

Introdução

- Criado por Tim Peters em 2002
- Algoritmo de ordenação padrão na linguagem de programação *Python* (método .sort(), função sorted(), etc) até 2022

Introdução

- Criado por Tim Peters em 2002
- Algoritmo de ordenação padrão na linguagem de programação *Python* (método .sort(), função sorted(), etc) até 2022
- Híbrido, junta características do MergeSort e do InsertionSort

Introdução

- Criado por Tim Peters em 2002
- Algoritmo de ordenação padrão na linguagem de programação *Python* (método .sort(), função sorted(), etc) até 2022
- Híbrido, junta características do MergeSort e do InsertionSort
- Emprega algumas heurísticas para maior desempenho (galope, espaço auxiliar reduzido, etc)

Introdução

- Criado por Tim Peters em 2002
- Algoritmo de ordenação padrão na linguagem de programação *Python* (método .sort(), função sorted(), etc) até 2022
- Híbrido, junta características do MergeSort e do InsertionSort
- Emprega algumas heurísticas para maior desempenho (galope, espaço auxiliar reduzido, etc)
- Usufrui da "ordem natural" de subarranjos para ordenar mais rapidamente

Introdução

- Criado por Tim Peters em 2002
- Algoritmo de ordenação padrão na linguagem de programação *Python* (método .sort(), função sorted(), etc) até 2022
- Híbrido, junta características do MergeSort e do InsertionSort
- Emprega algumas heurísticas para maior desempenho (galope, espaço auxiliar reduzido, etc)
- Usufrui da "ordem natural" de subarranjos para ordenar mais rapidamente
- Estável

Algoritmo Principal - Alto Nível

1. Divida o arranjo de entrada em *segmentos*

Algoritmo Principal - Alto Nível

1. Divida o arranjo de entrada em *segmentos*
2. Segmentos: subarranjos não-decrescentes ou estritamente decrescentes

Algoritmo Principal - Alto Nível

1. Divida o arranjo de entrada em *segmentos*
2. Segmentos: subarranjos não-decrescentes ou estritamente decrescentes
3. Cada segmento decrescente tem a ordem de seus elementos invertida

Algoritmo Principal - Alto Nível

1. Divida o arranjo de entrada em *segmentos*
2. Segmentos: subarranjos não-decrescentes ou estritamente decrescentes
3. Cada segmento decrescente tem a ordem de seus elementos invertida
4. Segmentos adjacentes são mesclados como no MergeSort

Algoritmo Principal - Alto Nível

1. Divida o arranjo de entrada em *segmentos*
2. Segmentos: subarranjos não-decrescentes ou estritamente decrescentes
3. Cada segmento decrescente tem a ordem de seus elementos invertida
4. Segmentos adjacentes são mesclados como no MergeSort
5. Execute o passo 4 até que reste apenas um segmento

Subrotinas em Alto Nível

- O cálculo de *minseg*, o tamanho mínimo de segmento, é feito através de uma manipulação com os *bits* que representam o tamanho do arranjo

Subrotinas em Alto Nível

- O cálculo de *minseg*, o tamanho mínimo de segmento, é feito através de uma manipulação com os *bits* que representam o tamanho do arranjo
- A fase de segmentação delimita os segmentos do arranjo e os empilha em uma pilha que será usada na fase de colapso

Subrotinas em Alto Nível

- O cálculo de *minseg*, o tamanho mínimo de segmento, é feito através de uma manipulação com os *bits* que representam o tamanho do arranjo
- A fase de segmentação delimita os segmentos do arranjo e os empilha em uma pilha que será usada na fase de colapso
- A subrotina de mescla é similar a do MergeSort, com a diferença principal sendo na redução de consumo de memória auxiliar e na integração de um algoritmo chamado de galope

Subrotinas em Alto Nível

- O cálculo de *minseg*, o tamanho mínimo de segmento, é feito através de uma manipulação com os *bits* que representam o tamanho do arranjo
- A fase de segmentação delimita os segmentos do arranjo e os empilha em uma pilha que será usada na fase de colapso
- A subrotina de mescla é similar a do MergeSort, com a diferença principal sendo na redução de consumo de memória auxiliar e na integração de um algoritmo chamado de galope
  - O consumo de memória auxiliar é reduzida graças a uma técnica inteligente que utiliza busca binária

Subrotinas em Alto Nível

- O cálculo de *minseg*, o tamanho mínimo de segmento, é feito através de uma manipulação com os *bits* que representam o tamanho do arranjo
- A fase de segmentação delimita os segmentos do arranjo e os empilha em uma pilha que será usada na fase de colapso
- A subrotina de mescla é similar a do MergeSort, com a diferença principal sendo na redução de consumo de memória auxiliar e na integração de um algoritmo chamado de galope
  - O consumo de memória auxiliar é reduzida graças a uma técnica inteligente que utiliza busca binária
  - O galope é uma subrotina feita com uma heurística baseada na ordem "natural" dos elementos do arranjo

Subrotinas em Alto Nível

- O cálculo de *minseg*, o tamanho mínimo de segmento, é feito através de uma manipulação com os *bits* que representam o tamanho do arranjo
- A fase de segmentação delimita os segmentos do arranjo e os empilha em uma pilha que será usada na fase de colapso
- A subrotina de mescla é similar a do MergeSort, com a diferença principal sendo na redução de consumo de memória auxiliar e na integração de um algoritmo chamado de galope
  - O consumo de memória auxiliar é reduzida graças a uma técnica inteligente que utiliza busca binária
  - O galope é uma subrotina feita com uma heurística baseada na ordem "natural" dos elementos do arranjo
    - Pode reduzir comparações entre elementos

Subrotinas em Alto Nível

- O cálculo de *minseg*, o tamanho mínimo de segmento, é feito através de uma manipulação com os *bits* que representam o tamanho do arranjo
- A fase de segmentação delimita os segmentos do arranjo e os empilha em uma pilha que será usada na fase de colapso
- A subrotina de mescla é similar a do MergeSort, com a diferença principal sendo na redução de consumo de memória auxiliar e na integração de um algoritmo chamado de galope
  - O consumo de memória auxiliar é reduzida graças a uma técnica inteligente que utiliza busca binária
  - O galope é uma subrotina feita com uma heurística baseada na ordem "natural" dos elementos do arranjo
    - Pode reduzir comparações entre elementos
- A ordem das mesclas é regida por invariantes que tentam garantir que o TimSort execute rapidamente e consuma pouco espaço

Sumário

Tamanho Mínimo de Segmento

Os segmentos devem ter um tamanho mínimo, denotado aqui por *minseg*.

$$minseg = \min \left\{ k \in \mathbb{N} : \left\lceil \frac{n}{k} \right\rceil = 2^m \right\}, \; k \in [32..64], \; m \in \mathbb{N}_{\geq 1}$$

Onde n é o tamanho do arranjo.

- A relação do teto aqui implica que cada segmento deve ter, no mínimo, tamanho igual ou ligeiramente menor que uma potência de dois
- Cada segmento deve ter ao menos dois elementos, senão é impossível determinar se este será não-decrescente ou estritamente decrescente

Quando dois segmentos têm tamanho aproximado ou exatamente igual, é dito que eles são *equilibrados*.

$$A = [1, 2, 3, 4]$$

$$B = [5, 6, 7, 8]$$

Figura: Exemplo de segmentos equilibrados

- Segmentos equilibrados = menos operações no processo de mesclagem

minseg

- Quando o algoritmo encontra um segmento de tamanho menor que *minseg*, este é estendido através de uma variação do InsertionSort com busca binária
- Elementos subsequentes em relação ao segmento são inseridos nele de forma ordenada até que o tamanho deste seja *minseg*
- O tamanho dos segmentos "pequenos" ficará alinhado a *minseg*, portanto fazendo deles equilibrados
- Segmentos pouco ordenados normalmente não irão gerar sequências ordenadas muito grandes, então essa abordagem é interessante para esse caso

minseg

Agora se tratando do cálculo de *minseg*, é possível obter o valor através de um algoritmo, formulado por Peters:

CalculaMinseg(n)

Entrada: $n \in \mathbb{N}$ tal que n é o tamanho de um arranjo a ser segmentado pelo TimSort
Saída : O tamanho mínimo de segmento *minseg*
$b \leftarrow$ seis bits mais significativos de n
$n' \leftarrow n$ sem os seis bits mais significativos
Se $b_{n'} > 0$
    $b \leftarrow b + 1$
Converta b para base 10
Devolva b

Sumário

Segmentação

Segmenta(A)

Entrada: Um arranjo A
Saída : Uma pilha $\mathcal{S}$ com os segmentos de A
$\mathcal{S} \leftarrow$ pilha vazia
Se $|A| < 64$
    InsertionSort$(A, 1, |A|)$
    Empilhe A em $\mathcal{S}$
Senão
    minseg $\leftarrow$ CalculaMinseg$(|A|)$
    Enquanto *houverem elementos a serem considerados*
        $s \leftarrow$ próximo segmento
        Enquanto $|s| <$ *minseg*
            Insira elementos em s com InsertionSort
        Empilhe s em $\mathcal{S}$
Devolva $\mathcal{S}$

Sumário

Algoritmo de Mescla

Considere que existem dois segmentos a serem mesclados. O algoritmo de mescla funciona da seguinte maneira:

1. Comece da mesma forma que uma mescla linear (MergeSort)

[1] Isto é, se forem selecionados no processo de mesclagem

Algoritmo de Mescla

Considere que existem dois segmentos a serem mesclados. O algoritmo de mescla funciona da seguinte maneira:

1. Comece da mesma forma que uma mescla linear (MergeSort)
2. Se *mingalope* elementos "ganharem"[1] consecutivamente em algum dos segmentos, execute a mescla com galope para eles

[1] Isto é, se forem selecionados no processo de mesclagem

Algoritmo de Mescla

Considere que existem dois segmentos a serem mesclados. O algoritmo de mescla funciona da seguinte maneira:

1. Comece da mesma forma que uma mescla linear (MergeSort)
2. Se *mingalope* elementos "ganharem"[1] consecutivamente em algum dos segmentos, execute a mescla com galope para eles
3. Se a mescla com galope parar, volte à mescla linear

[1] Isto é, se forem selecionados no processo de mesclagem

Algoritmo de Mescla

Considere que existem dois segmentos a serem mesclados. O algoritmo de mescla funciona da seguinte maneira:

1. Comece da mesma forma que uma mescla linear (MergeSort)
2. Se *mingalope* elementos "ganharem"[1] consecutivamente em algum dos segmentos, execute a mescla com galope para eles
3. Se a mescla com galope parar, volte à mescla linear

- Segundo testes empíricos feitos por Peters, um bom valor para *mingalope* é 8

[1] Isto é, se forem selecionados no processo de mesclagem

Mescla com Galope

- Estratégia usada para quando muitos elementos de um segmento são copiados para o segmento final
- Heurística: "Se vários elementos de um segmento foram selecionados consecutivamente, provavelmente mais serão"
- Consiste em uma espécie de "aposta", na qual se assume que os elementos de um segmento frequentemente selecionado são, em grande parte, menores que os do outro

Mescla com Galope

Considere que A e B são segmentos a serem mesclados, e que A "ativou" a mescla com galope.

1. Busque $A[1]$ em B via busca exponencial

Mescla com Galope

Considere que A e B são segmentos a serem mesclados, e que A "ativou" a mescla com galope.

1. Busque $A[1]$ em B via busca exponencial
2. Se o índice de $A[1]$ em B é k, copie todos os elementos de $B[1]$ até $B[k-1]$ para o segmento final

Mescla com Galope

Considere que A e B são segmentos a serem mesclados, e que A "ativou" a mescla com galope.

1. Busque $A[1]$ em B via busca exponencial
2. Se o índice de $A[1]$ em B é k, copie todos os elementos de $B[1]$ até $B[k-1]$ para o segmento final
3. Desconsidere os elementos copiados

Mescla com Galope

Considere que A e B são segmentos a serem mesclados, e que A "ativou" a mescla com galope.

1. Busque $A[1]$ em B via busca exponencial
2. Se o índice de $A[1]$ em B é k, copie todos os elementos de $B[1]$ até $B[k-1]$ para o segmento final
3. Desconsidere os elementos copiados
4. Copie $A[1]$ para o segmento final

Mescla com Galope

Considere que A e B são segmentos a serem mesclados, e que A "ativou" a mescla com galope.

1. Busque $A[1]$ em B via busca exponencial
2. Se o índice de $A[1]$ em B é k, copie todos os elementos de $B[1]$ até $B[k-1]$ para o segmento final
3. Desconsidere os elementos copiados
4. Copie $A[1]$ para o segmento final
5. Realize o processo de forma similar, mas buscando $B[1]$ em A

Mescla com Galope

Considere que A e B são segmentos a serem mesclados, e que A "ativou" a mescla com galope.

1. Busque $A[1]$ em B via busca exponencial
2. Se o índice de $A[1]$ em B é k, copie todos os elementos de $B[1]$ até $B[k-1]$ para o segmento final
3. Desconsidere os elementos copiados
4. Copie $A[1]$ para o segmento final
5. Realize o processo de forma similar, mas buscando $B[1]$ em A
6. Se o tamanho de ambos os segmentos copiados for menor que *mingalope*, pare a mescla com galope

Mescla com Galope

Considere que A e B são segmentos a serem mesclados, e que A "ativou" a mescla com galope.

1. Busque $A[1]$ em B via busca exponencial
2. Se o índice de $A[1]$ em B é k, copie todos os elementos de $B[1]$ até $B[k-1]$ para o segmento final
3. Desconsidere os elementos copiados
4. Copie $A[1]$ para o segmento final
5. Realize o processo de forma similar, mas buscando $B[1]$ em A
6. Se o tamanho de ambos os segmentos copiados for menor que *mingalope*, pare a mescla com galope
7. Repita os passos anteriores até que um dos segmentos tenha tido todos os seus elementos copiados para o segmento final

Mescla com Galope

Considere que A e B são segmentos a serem mesclados, e que A "ativou" a mescla com galope.

1. Busque $A[1]$ em B via busca exponencial
2. Se o índice de $A[1]$ em B é k, copie todos os elementos de $B[1]$ até $B[k-1]$ para o segmento final
3. Desconsidere os elementos copiados
4. Copie $A[1]$ para o segmento final
5. Realize o processo de forma similar, mas buscando $B[1]$ em A
6. Se o tamanho de ambos os segmentos copiados for menor que *mingalope*, pare a mescla com galope
7. Repita os passos anteriores até que um dos segmentos tenha tido todos os seus elementos copiados para o segmento final
8. Copie, para o segmento final, o restante dos elementos do segmento que ainda não foi completamente copiado

Mescla com Galope

Considere que A e B são segmentos a serem mesclados, e que A "ativou" a mescla com galope.

1. Busque $A[1]$ em B via busca exponencial
2. Se o índice de $A[1]$ em B é k, copie todos os elementos de $B[1]$ até $B[k-1]$ para o segmento final
3. Desconsidere os elementos copiados
4. Copie $A[1]$ para o segmento final
5. Realize o processo de forma similar, mas buscando $B[1]$ em A
6. Se o tamanho de ambos os segmentos copiados for menor que *mingalope*, pare a mescla com galope
7. Repita os passos anteriores até que um dos segmentos tenha tido todos os seus elementos copiados para o segmento final
8. Copie, para o segmento final, o restante dos elementos do segmento que ainda não foi completamente copiado

- Para segmentos menores que *mingalope*, a mescla com galope geralmente é mais custosa que uma mescla linear [Peters, 2002]

Busca Exponencial

- Algoritmo de busca para arranjos ordenados
- Funciona dobrando o índice de busca até que se ache um intervalo onde o elemento possivelmente está
- Executa busca binária no intervalo
- Pior caso $\mathcal{O}(\log i)$, onde i é o índice do elemento sendo buscado

Busca Exponencial

BuscaExponencial(A, x)

Entrada: Um arranjo ordenado A e um elemento x a ser buscado em A

Saída : A posição de x em A

Se $|A| = 0$

    Devolva *NAO_ACHOU*

$limite \leftarrow 1$

Enquanto $limite < |A|$ *e* $A[limite] < x$

    $limite \leftarrow 2 * limite$

Devolva *BuscaBinaria* $\left(A, \left\lfloor \frac{limite}{2} \right\rfloor, \min\{limite, |A|\}\right)$

- Intervalo para a busca binária: $[2^{k-1}.. \min\{2^k, |A|\}]$
- k é o número de vezes que o limite foi dobrado no algoritmo

Economia de Memória Auxiliar

Sejam A e B dois segmentos a serem mesclados. O TimSort utiliza de uma técnica para economizar memória auxiliar, com base na ordem dos elementos em A e B, em forma de algoritmo:

1. Busque $A[|A|]$ em B e $B[1]$ em A via busca binária

[2] $A[i] = B[1]$ e $B[j] = A[|A|]$

[3] Os elementos em C correspondentes a $A[i+1..|A|]$ e $B[1..j-1]$

Economia de Memória Auxiliar

Sejam A e B dois segmentos a serem mesclados. O TimSort utiliza de uma técnica para economizar memória auxiliar, com base na ordem dos elementos em A e B, em forma de algoritmo:

1. Busque $A[|A|]$ em B e $B[1]$ em A via busca binária
2. Copie o subarranjo de menor tamanho entre $A[i+1..|A|]$ e $B[1..j-1]$[2] para um *buffer* auxiliar

[2] $A[i] = B[1]$ e $B[j] = A[|A|]$
[3] Os elementos em C correspondentes a $A[i+1..|A|]$ e $B[1..j-1]$

Economia de Memória Auxiliar

Sejam A e B dois segmentos a serem mesclados. O TimSort utiliza de uma técnica para economizar memória auxiliar, com base na ordem dos elementos em A e B, em forma de algoritmo:

1. Busque $A[|A|]$ em B e $B[1]$ em A via busca binária
2. Copie o subarranjo de menor tamanho entre $A[i+1..|A|]$ e $B[1..j-1]$[2] para um *buffer* auxiliar
3. Combine A e B em um só espaço de memória, obtendo C

[2] $A[i] = B[1]$ e $B[j] = A[|A|]$

[3] Os elementos em C correspondentes a $A[i+1..|A|]$ e $B[1..j-1]$

Economia de Memória Auxiliar

Sejam A e B dois segmentos a serem mesclados. O TimSort utiliza de uma técnica para economizar memória auxiliar, com base na ordem dos elementos em A e B, em forma de algoritmo:

1. Busque $A[|A|]$ em B e $B[1]$ em A via busca binária
2. Copie o subarranjo de menor tamanho entre $A[i+1..|A|]$ e $B[1..j-1]$[2] para um *buffer* auxiliar
3. Combine A e B em um só espaço de memória, obtendo C
4. Realize a mescla entre a parte copiada para o *buffer* e a que não foi, sobrescrevendo os elementos de $A+B$ no intervalo de interesse[3]

[2] $A[i] = B[1]$ e $B[j] = A[|A|]$

[3] Os elementos em C correspondentes a $A[i+1..|A|]$ e $B[1..j-1]$

Economia de Memória Auxiliar

Sejam A e B dois segmentos a serem mesclados. O TimSort utiliza de uma técnica para economizar memória auxiliar, com base na ordem dos elementos em A e B, em forma de algoritmo:

1. Busque $A[|A|]$ em B e $B[1]$ em A via busca binária
2. Copie o subarranjo de menor tamanho entre $A[i+1..|A|]$ e $B[1..j-1]$[2] para um *buffer* auxiliar
3. Combine A e B em um só espaço de memória, obtendo C
4. Realize a mescla entre a parte copiada para o *buffer* e a que não foi, sobrescrevendo os elementos de $A+B$ no intervalo de interesse[3]

- Note que os elementos em $A[1..i]$ e $B[j..|B|]$ já estão nas posições corretas, em relação ao segmento após a mescla

[2] $A[i] = B[1]$ e $B[j] = A[|A|]$

[3] Os elementos em C correspondentes a $A[i+1..|A|]$ e $B[1..j-1]$

Economia de Memória Auxiliar

A mescla ocorre bidirecionalmente, isto é:

- Se A foi copiado para o *buffer*, realize a mescla da esquerda para a direita (como no MergeSort) entre o *buffer* e B
- Se B foi copiado, realize a mescla da direita para a esquerda entre o *buffer* e A

Esta abordagem previne que elementos sejam perdidos na mescla, caso esta ocorresse em apenas uma direção.

Colapso de Pilha

Invariantes da Pilha de Mescla [Auger et al., 2018, de Gouw et al., 2015]

Sejam $\mathcal{P}$ uma pilha de segmentos a serem mesclados e $\mathcal{P}[1], \mathcal{P}[2], \mathcal{P}[3]$ e $\mathcal{P}[4]$ segmentos próximos ao topo de $\mathcal{P}$, com $\mathcal{P}[1]$ sendo o elemento do topo e $\mathcal{P}[4]$ sendo o mais distante do topo dentre estes.
Os invariantes mantidos em $\mathcal{P}$ são os seguintes:

1. $|\mathcal{P}[3]| \geq |\mathcal{P}[1]|$
2. $|\mathcal{P}[3]| > |\mathcal{P}[2]| + |\mathcal{P}[1]|$
3. $|\mathcal{P}[4]| > |\mathcal{P}[3]| + |\mathcal{P}[2]|$
4. $|\mathcal{P}[2]| > |\mathcal{P}[1]|$

- O tamanho de cada segmento em $\mathcal{P}$ cresce pelo menos tão rápido quanto os números de Fibonacci [Peters, 2002, Auger et al., 2018]
- Consequentemente, o número de segmentos em $\mathcal{P}$ é $\mathcal{O}(\log n)$, para um arranjo de n elementos

Colapso de Pilha

- Se violado, o invariante 1 é restaurado ao mesclar $\mathcal{P}[3]$ com $\mathcal{P}[2]$
- Se violados, os demais são restaurados ao mesclar $\mathcal{P}[2]$ com $\mathcal{P}[1]$
- Apenas segmentos adjacentes são mesclados (a estabilidade do algoritmo é mantida [Peters, 2002])
- Todo esse processo de manutenção de invariantes é chamado de "colapso de pilha"

Colapso de Pilha

Colapsa($\mathcal{P}$)

Entrada: Uma pilha de segmentos $\mathcal{P}$

Saída : $\mathcal{P}$ modificada de forma que satisfaça os invariantes da pilha de mescla

Enquanto $|\mathcal{P}| > 1$

    $n \leftarrow |\mathcal{P}| - 2$

    Se $n > 0$ *e* $|\mathcal{P}[3]| \leq |\mathcal{P}[2]| + |\mathcal{P}[1]|$ *ou* $n > 1$ *e* $|\mathcal{P}[4]| \leq |\mathcal{P}[3]| + |\mathcal{P}[2]|$

        Se $|\mathcal{P}[3] < |\mathcal{P}[1]|$

            Mescla($\mathcal{P}[2]$, $\mathcal{P}[3]$)

        Senão

            Mescla($\mathcal{P}[1]$, $\mathcal{P}[2]$)

    Senão Se $|\mathcal{P}[2]| \leq |\mathcal{P}[1]|$

        Mescla($\mathcal{P}[1], \mathcal{P}[2]$)

    Senão

        break

Colapso de Pilha

O intuito do colapso de pilha é fazer com que:

$$\begin{cases} |\mathcal{P}[i+2]| > |\mathcal{P}[i+1]| + |\mathcal{P}[i]| \\ |\mathcal{P}[i+1]| > |\mathcal{P}[i]| \end{cases}$$

- $\forall i \in [1..|A|-2]$

Sumário

Algoritmo Principal

TimSort(A)

Entrada: Um arranjo A
Saída : O arranjo A, ordenado
$\mathcal{S} \leftarrow$ Segmenta(A)
$\mathcal{P} \leftarrow$ pilha vazia
Enquanto $\mathcal{S}$ *não está vazia*
    Desempilhe um segmento de $\mathcal{S}$ e empilhe-o em $\mathcal{P}$
    Enquanto *algum invariante não estiver satisfeito por* $\mathcal{P}$
        Colapsa($\mathcal{P}$)
Enquanto $|\mathcal{P}| > 1$
    Mescla($\mathcal{P}[1], \mathcal{P}[2]$)
Devolva $\mathcal{P}[1]$

Sumário

TimSort e *Cache*

O TimSort faz bom uso dos princípios de localidade espacial e de localidade temporal:

- A extensão de segmentos pequenos acessa apenas elementos próximos (subsequentes)
- Apenas segmentos adjacentes são acessados no colapso de pilha
- A mescla com galope move elementos adjacentes em blocos
- A busca exponencial usada na mescla com galope acessa poucos elementos não adjacentes
- O intervalo delimitado na busca exponencial provavelmente já está na *cache* antes de realizar a busca binária nele
- Os invariantes do colapso de pilha fazem com que, na maioria das vezes, segmentos acessados recentemente[4] sejam mesclados

[4] Isto é, segmentos próximos ao topo da pilha de segmentos

Sumário

O *Bug* do TimSort

- de Gouw et al. publicaram um artigo sobre uma falha crítica no TimSort
- A falha era, em suma, um *stack overflow*
- No contexto da linguagem no qual foi descoberto (*Java*), era um erro `ArrayIndexOutOfBoundsException`
- Ocorria na parte do colapso de $\mathcal{P}$, com esta sendo a mesma pilha do algoritmo principal
- A versão original do TimSort não mantinha invariantes para os quatro primeiros segmentos de $\mathcal{P}$, mas sim para três

O *Bug* do TimSort

O tamanho pré-alocado da pilha de segmentos era baseada no predicado abaixo:

$Pr(i)$

$Pr(i) :=$ As condições descritas no slide 25 são mantidas, sendo $\mathcal{P}$ a mesma pilha mencionada no algoritmo principal do TimSort.

Invariantes do TimSort Original

Os invariantes que a versão original do TimSort mantinha eram os seguintes:

1. $|\mathcal{P}[3]| > |\mathcal{P}[2]| + |\mathcal{P}[1]|$
2. $|\mathcal{P}[2]| > |\mathcal{P}[1]|$

- O invariante 1 era restaurado mesclando $\mathcal{P}[2]$ com o menor segmento entre $\mathcal{P}[3]$ e $\mathcal{P}[1]$
- O invariante 2 era restaurado mesclando $\mathcal{P}[2]$ com $\mathcal{P}[1]$

O *Bug* do TimSort

Mas qual o problema?

- $Pr(i)$ não era mantido para qualquer configuração de $\mathcal{P}$
- Motivo: Os invariantes para os três primeiros elementos da pilha não eram suficientes para manter $Pr(i)$

Seja (A, B, C, D) uma configuração de $\mathcal{P}$. D é o topo de $\mathcal{P}$.

- Suponha que essa configuração satisfaz $Pr(i)$, isto é: $|A| > |B| + |C| + |D|$, $|B| > |C| + |D|$ e $|C| > |D|$

O *Bug* do TimSort

Mas qual o problema?

- $Pr(i)$ não era mantido para qualquer configuração de $\mathcal{P}$
- Motivo: Os invariantes para os três primeiros elementos da pilha não eram suficientes para manter $Pr(i)$

Seja (A, B, C, D) uma configuração de $\mathcal{P}$. D é o topo de $\mathcal{P}$.

- Suponha que essa configuração satisfaz $Pr(i)$, isto é: $|A| > |B| + |C| + |D|$, $|B| > |C| + |D|$ e $|C| > |D|$
- Se um segmento E for inserido em $\mathcal{P}$, esta agora será (A, B, C, D, E)

O *Bug* do TimSort

Mas qual o problema?

- $Pr(i)$ não era mantido para qualquer configuração de $\mathcal{P}$
- Motivo: Os invariantes para os três primeiros elementos da pilha não eram suficientes para manter $Pr(i)$

Seja (A, B, C, D) uma configuração de $\mathcal{P}$. D é o topo de $\mathcal{P}$.

- Suponha que essa configuração satisfaz $Pr(i)$, isto é: $|A| > |B| + |C| + |D|$, $|B| > |C| + |D|$ e $|C| > |D|$
- Se um segmento E for inserido em $\mathcal{P}$, esta agora será (A, B, C, D, E)
- Se $|C| \leq |D| + |E|$ e $|C| < |E|$, então C e D serão mesclados, resultando na configuração $(A, B, C + D, E)$

O *Bug* do TimSort

Mas qual o problema?

- $Pr(i)$ não era mantido para qualquer configuração de $\mathcal{P}$
- Motivo: Os invariantes para os três primeiros elementos da pilha não eram suficientes para manter $Pr(i)$

Seja (A, B, C, D) uma configuração de $\mathcal{P}$. D é o topo de $\mathcal{P}$.

- Suponha que essa configuração satisfaz $Pr(i)$, isto é: $|A| > |B| + |C| + |D|$, $|B| > |C| + |D|$ e $|C| > |D|$
- Se um segmento E for inserido em $\mathcal{P}$, esta agora será (A, B, C, D, E)
- Se $|C| \leq |D| + |E|$ e $|C| < |E|$, então C e D serão mesclados, resultando na configuração $(A, B, C + D, E)$
- Os invariantes serão checados para $B, C + D$ e E e serão restaurados caso violados

O *Bug* do TimSort

Mas qual o problema?

- $Pr(i)$ não era mantido para qualquer configuração de $\mathcal{P}$
- Motivo: Os invariantes para os três primeiros elementos da pilha não eram suficientes para manter $Pr(i)$

Seja (A, B, C, D) uma configuração de $\mathcal{P}$. D é o topo de $\mathcal{P}$.

- Suponha que essa configuração satisfaz $Pr(i)$, isto é: $|A| > |B| + |C| + |D|$, $|B| > |C| + |D|$ e $|C| > |D|$
- Se um segmento E for inserido em $\mathcal{P}$, esta agora será (A, B, C, D, E)
- Se $|C| \leq |D| + |E|$ e $|C| < |E|$, então C e D serão mesclados, resultando na configuração $(A, B, C + D, E)$
- Os invariantes serão checados para $B, C + D$ e E e serão restaurados caso violados
- Nada garante que $|A| > |B| + |C + D| + |E|$, pois A é excluído da verificação

O *Bug* do TimSort

Mas qual o problema?

- $Pr(i)$ não era mantido para qualquer configuração de $\mathcal{P}$
- Motivo: Os invariantes para os três primeiros elementos da pilha não eram suficientes para manter $Pr(i)$

Seja (A, B, C, D) uma configuração de $\mathcal{P}$. D é o topo de $\mathcal{P}$.

- Suponha que essa configuração satisfaz $Pr(i)$, isto é: $|A| > |B| + |C| + |D|$, $|B| > |C| + |D|$ e $|C| > |D|$
- Se um segmento E for inserido em $\mathcal{P}$, esta agora será (A, B, C, D, E)
- Se $|C| \leq |D| + |E|$ e $|C| < |E|$, então C e D serão mesclados, resultando na configuração $(A, B, C + D, E)$
- Os invariantes serão checados para $B, C + D$ e E e serão restaurados caso violados
- Nada garante que $|A| > |B| + |C + D| + |E|$, pois A é excluído da verificação
- Conclusão: $(A, B, C + D, E)$ pode não satisfazer $Pr(i)$

O *Bug* do TimSort

- de Gouw et al. testaram para um arranjo de tamanho 67108864
- O tamanho pré-alocado não era suficiente, ocasionando um *stack overflow*
- Os autores corrigiram o TimSort, fazendo com que a subrotina de colapso considerasse invariantes para os quatro primeiros elementos de $\mathcal{P}$, em vez de três
- Nesta apresentação, já é considerada a versão "definitiva" do TimSort, como se pode ver no slide 22
- Com essa correção, $\mathcal{P}$ satisfaz $Pr(i)$ [de Gouw et al., 2015]
- O *bug* também ocorria na implementação em *Python*, pois era similar
- Em ambas as implementações, o *bug* foi corrigido
- O *Java* continuou com o TimSort
- O *Python* migrou para o PowerSort em 2022 [James, 2022]

Sumário

Conclusões

Podemos concluir algumas coisas desse estudo:

- O TimSort é um algoritmo que tem um bom desempenho em dados parcialmente ordenados
- Contorna bem situações na qual os dados possuem pouca ordem, com a extensão de segmentos
- Usa heurísticas geralmente efetivas para economizar comparações, movimentações de dados e memória auxiliar
- Faz bom uso da *cache*
- Se o arranjo estiver ordenado, o TimSort é de ordem $\mathcal{O}(n)$, uma vez que todo o arranjo será um único segmento, para um arranjo de n elementos
- Pior caso do algoritmo é $\mathcal{O}(n \log n)$
- A prova formal da complexidade de pior caso se encontra em [Auger et al., 2018], a qual iremos abordar em um seminário futuro

Questionamentos

Algumas questões ficam no ar...

- Lendo a descrição de Peters, é possível perceber que alguns parâmetros[5] foram escolhidos, em grande parte, por testes empíricos
- Será que existe alguma entrada específica, a qual Peters não testou, que atenua o desempenho do algoritmo na prática?
- Será que o galope não seria melhor se começasse a busca pelo primeiro elemento do segmento que "perdeu", em vez do que "ganhou"?

[5] *mingalope*, *minseg*, tamanho mínimo para segmentação (64)

Sumário

Referências I

Nicolas Auger, Vincent Jugé, Cyril Nicaud, and Carine Pivoteau. On the Worst-Case Complexity of TimSort. In Yossi Azar, Hannah Bast, and Grzegorz Herman, editors, *26th Annual European Symposium on Algorithms (ESA 2018)*, volume 112 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 4:1–4:13, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-95977-081-1. doi: 10.4230/LIPIcs.ESA.2018.4. URL `http://drops.dagstuhl.de/opus/volltexte/2018/9467`.

Stijn de Gouw, Jurriaan Rot, Frank S. de Boer, Richard Bubel, and Reiner Hähnle. Openjdk's java.utils.collection.sort() is broken: The good, the bad and the worst case. In Daniel Kroening and Corina S. Păsăreanu, editors, *Computer Aided Verification*, pages 273–289, Cham, 2015. Springer International Publishing. ISBN 978-3-319-21690-4. doi: https://doi.org/10.1007/978-3-319-21690-4_16. URL `https://www.researchgate.net/publication/300646623_`

Referências II

OpenJDK's_JavautilsCollectionsort_Is_Broken_The_Good_the_Bad_and_the_Worst_Case.

Mike James. Python Now Uses Powersort, dec 2022. URL https://www.i-programmer.info/news/216-python/15954-python-now-uses-powersort.html. Acesso em: 12/06/2025.

Tim Peters. Descrição do TimSort. https://bugs.python.org/file4451/timsort.txt, 2002. Acesso em: 27/11/2025.