TESTE DE EFICIÊNCIA DE MÉTODOS DE ORDENAÇÃO DE DADOS USANDO A LINGUAGEM DE PROGRAMAÇÃO GO

Lucas Eduardo Wendt, Glória Patrícia Lopes

Bacharelado em Ciência da Computação, UTFPR Campus Santa Helena

lucaswendt@alunos.utfpr.edu.br, gloriap@utfpr.edu.br

**RESUMO**

Este trabalho é um teste comparativo com algoritmos de ordenação, como: *bubble*, *heap*, *insertion*, *merge*, *quick*, *radix*, *selection* e *shell sort*. Os algoritmos são implementados utilizando a linguagem Go de forma que o tempo seja o mais próximo possível do tempo de execução do algoritmo propriamente dito. O programa é construído de forma que este inicia o próximo passo do teste apenas quando a máquina satisfaça um critério de temperatura predefinido descartando erro humano do processo de teste propriamente dito.

**Palavras-chave**: Algoritmos de Ordenação; Benchmark; Golang; Pesquisa; Ordenação de Dados.

***ABSTRACT***

*This paper is a comparative test with sorting algorithms, like: bubble, heap, insertion, merge, quick, radix, selection and shell sort. The algorithms were implemented using the Go programming language so that the time be the nearest to the run time of the algorithm itself. The program is built in a form that it starts the next step only when the machine satisfies a temperature criteria discarding human error in the process of the test itself.*

***Keywords****: Sorting Algorithms; Benchmark; Golang; Data Search and Data Sort.*

**INTRODUÇÃO**

Atualmente se demonstra muito importante manter a organização para produzir melhor. Com máquinas não é diferente, estas podem trabalhar de forma mais eficiente com dados organizados, mas esta utopia nem sempre acontece, e quando os dados não estão em ordem é necessário ordena-los e organizá-los. A missão de organizar um de muitos tipos de dados, que são vetores de valores comparáveis, é delegada para algoritmos de ordenação. Este trabalho tem como objetivo comparar a eficiência dos principais métodos de ordenação de dados aplicados a vetores de números inteiros implementando-os e comparando o tempo utilizado para ordenação de vetores de diferentes tamanhos com valores numéricos aleatórios. Os algoritmos testados são o *bubble, heap, insertion, merge, quick, radix, selection* e *shell sort*. As implementações, feitas na linguagem Go, são as variações mais simples de cada algoritmo, suficientes para ordenar o vetor com sucesso.

# REFERENCIAL TEÓRICO-METODOLÓGICO

* 1. REFERENCIAL TEÓRICO

Diversos artigos já realizaram testes comparativos com algoritmos de ordenação, como o de Garcia (2014) e o de Francis e Mathielson (1988) utilizando as linguagens Java e C respectivamente e utilizando implementações multiprocessadas.

O trabalho de Francis e Mathielson (1988) é sobre o “primeiro algoritmo de ordenação paralelo para multiprocessadores múltipla instrução múltiplos dados de memória compartilhada”. Por ser um algoritmo baseado no *merge sort* com as subdivisões o limite de núcleos de processador envolvidos depende do tamanho do vetor. Segundo o artigo, sendo *n* o tamanho do vetor e *p* a quantidade de processadores envolvidos tem uma complexidade de tempo igual a ) e uma complexidade de espaço igual a .

O algoritmo é mais aplicável onde . Logo é mais indicado onde o vetor seja menor que o número de processadores envolvidos.

Segundo Garcia (2014), a complexidade de tempo, para fins de comparação, do *merge sort* monoprocessado é:

Segundo Garcia (2014), a eficiência dos algoritmos de ordenação pode estar relacionada a vários fatores, tais como: o tamanho da entrada de dados, quantidade de memória disponível, entre outros.

### Bubble Sort

Segundo a definição de Min (2010), uma variável auxiliar, a qual é chamada de *exchanged* pode ser definida e utilizada para determinar se houve movimentação dos elementos do vetor durante a última volta de ordenação. No início, esta variável é definida como 0, ou seja, falso. A variável é então alterada para 1, ou verdadeiro, assim que é realizada uma operação de troca de elementos. Se ao final da iteração este valor não foi alterado para verdadeiro então o vetor está ordenado e o algoritmo concluído.

### Heap Sort

Segundo Schaffer e Sedgewick (1993), o *heap sort* trata o *array*, ou vetor, como uma árvore completa armazenada em ordem de nível e é baseado em manter uma ordem de *heap* nos valores do vetor, definida como: Um heap de *N*chaves em um array tem maior que e para . Equivalentemente, é menos que para .

### Insertion Sort

Segundo Dutta (2013), o *insertion sort* funciona considerando inicialmente um vetor com elementos. Este algoritmo percorre o vetor do até o , inserindo cada elemento na sua posição apropriada no vetor previamente ordenado , , ..., . Isto pode ser realizado comparando com , e assim por diante, até achar um elemento de tal modo que . Então cada um dos elementos , ,... , é movido uma posição para frente, e é inserido na posição do vetor.

### Merge Sort

Segundo Qin, o *merge sort* usa uma abordagem dividir para conquistar. Primeiramente divide-se o vetor repetidamente em duas metades. Para-se de dividir quando sobrar apenas um elemento. A razão disto parte da premissa de que um vetor de apenas uma posição já está ordenado. Com os subvetores ordenados constrói-se o vetor final pegando o valor sempre do vetor que tem o menor menor valor até pegar todos os valores dos dois vetores. Junta-se os subvetores até ter-se o vetor completo novamente, porém, ordenado.

### Quick Sort

Segundo Hoare (1962), o *quick sort* é descrito como baseado no princípio de resolver um problema em outros dois problemas mais simples. Cada um destes subproblemas pode ser resolvido em problemas ainda mais simples. O processo é repetido até encontrar todos os problemas triviais. Estes problemas triviais podem então ser resolvidos por métodos conhecidos, deste modo obtendo a solução do problema mais complexo. Este processo de separar o problema em problemas menores é o particionamento.

### Radix Sort

Segundo Horsmalahti (2012), o *radix sort* é um algoritmo de ordenação de números inteiros que ordena todos os elementos usando um dígito por vez. Este algoritmo de ordenação era muito utilizado pelas máquinas de ordenar cartões. O vetor é primeiramente ordenado na base do maior digito. Depois disso, todos os cartões são ordenados utilizando o segundo maior digito e assim por diante. Existe a variação que vai do dígito mais significativo para o menos significativo e a que segue o caminho inverso.

### Shell Sort

Segundo Sedgewick (1996), o *shell sort* básico foi um dos primeiros algoritmos de ordenação a ser conhecido, criado por D. L. Shell em 1959 e, segundo o autor, está entre um dos métodos mais fáceis de se implementar. O algoritmo é baseado no *insertion sort*: percorre-se o vetor da esquerda para a direita inserindo cada elemento na posição entre os elementos da sua esquerda, que estão ordenados, movendo os valores maiores uma posição para a direita. O *shell sort* é uma sequência de *insertion sorts* intercalados baseados em uma sequência de incremento: para cada elemento , aplique *insertion sort* nos subvetores consistindo nos eiro, eiro, eiro, ... itens, para . Este processo é chamado pelo autor de *h-sorting*, ou ordenação h do vetor.

### Selection Sort

Segundo Jadoon, Solheria e Qayum, o *selection sort* é um dos algoritmos de ordenação mais fáceis e mais úteis para lidar com um quantidade pequena de dados. Apesar de realizar muitas comparações este método realiza o mínimo possível de movimentos nos dados. Portanto, se seus dados tem chaves pequenas mas são grandes em área de dados então o *selection* pode ser o mais rápido.

Segundo Chand, o *selection sort* é o algoritmo em que elementos sucessivos são selecionados em ordem e colocados em sua posição ordenada. O *selection sort* não usa espaço adicional para ordenar, exceto algumas variáveis temporárias. Em cada iteração o vetor é percorrido procurando o extremo, ao terminar de percorrer é realizada a troca mandando o valor no extremo em algum dos cantos. Se for realizado do maior para o menor valor o vetor será ordenado da direita para a esquerda e vice versa. Após iterações o vetor estará ordenado.

### A linguagem Go

Segundo Golang (2019), Go é uma linguagem de programação de propósito geral pensada para programação de sistemas em mente. É fortemente tipada, tem coletor de lixo, tem suporte explícito para programação concorrente. Esta suporta também o padrão Unicode nativamente sem precisar declarar explicitamente ou utilizar um compilador alternativo. Programas são construídos de pacotes, cujas propriedades permitem administração e compilação fácil e eficiente de dependências. A linguagem tem estruturas mais simplificadas, não tem classes como tradicionalmente se conhece. Os agrupamentos heterogêneos de dados são unicamente em structs, ou estruturas. A ideia é bem semelhante as estruturas do C, porém estas estruturas, e os tipos não primitivos em geral, tem métodos e interfaces definem máscaras para estruturas permitindo generalizações, como a *io.Writer*, a qual define uma forma universal na linguagem de se fazer escrita de dados em algum lugar. Se uma função recebe uma interface, a instancia da estrutura que é passada por parâmetro precisa obrigatoriamente possuir os métodos definidos na interface para ser aceito. A linguagem tem ponteiros e não existe a necessidade da notação “->” do C. O acesso dos membros acontece utilizando “.” em ambos os casos, seja ponteiro, seja o valor diretamente. A única estrutura de loop é o *for,* que pode ser utilizado passando uma expressão que retorne verdadeiro ou falso, a clássica sintaxe de inicialização, decremento e checagem utilizada, por exemplo, no C ou utilizando-se da palavra chave range para iterar sobre os vetores e maps. Por não ter a sintaxe try/catch, em Go as funções retornam os erros. Se a função retornar um erro nulo então ela foi concluída com sucesso. Checagens de erros utilizando estruturas condicionais são extremamente comuns em programas escritos na linguagem Go.

* 1. METODOLOGIA

O código do trabalho foi implementado usando a linguagem de programação Go.

Esta escolha começou com a curiosidade sobre a linguagem e o desejo de fazer alguma coisa interessante e útil com ela e por esta não ser muito explorada em meios acadêmicos. Este trabalho é uma tentativa de quebrar a monotonia dos artigos que geralmente utilizam linguagens mais populares, como C, C++ e Java.

Para realizar os testes será utilizada a biblioteca de testes e *benchmarks* da biblioteca padrão, ambos presentes no pacote *testing*. Foram-se utilizadas abordagens tirando proveito da funcionalidade de *clojures*, ou funções anônimas. Esta funcionalidade permite a aplicação da estratégia de construir a função no formato de teste ou *benchmark* que será chamada pela biblioteca de testes trazendo mais organização e reaproveitamento de código, além de reduzir repetição de código.

Os métodos são passados por uma suíte de testes unificada, utilizando a estratégia citada acima, a fim de garantir que estão realizando o processo corretamente e retornando uma saída confiável. Diversos casos de teste foram implementados, alguns com vetores pré-escritos e outros com valores aleatórios gerados na hora, a qual a sequência é testada por uma outra função. Esta função consiste em percorrer o vetor elemento por elemento comparando o elemento atual com o próximo, se o próximo elemento for menor que o atual esta função retorna falso indicando que o vetor não está ordenado.

Os dados passados no processo de *benchmark* da função são pré-aleatorizados em um arquivo de código Go. Este então é compilado junto com o programa principal garantindo o mesmo vetor em todas as iterações. Este arquivo de código Go tem aproximadamente 80MB.

Todo o código desenvolvido e os dados do resultado estarão presentes nos anexos.

Antes de cada operação de ordenação é realizada a cópia do vetor a ser ordenado para outra localização, isto se faz necessário em decorrência dos métodos receberem o vetor por referência, ou seja, eles alteram o dado passado como entrada, e em caso de mais de uma iteração no mesmo vetor por sessão isto pode ser desastroso, como métodos terminando prematuramente por já terem recebido o vetor ordenado.

O tempo contado em um *benchmark* consiste no tempo que o algoritmo leva para copiar o vetor e ordenar, ou seja, não é contado o tempo que o programa utiliza para inicializar, ler vetores, entre outras atividades que programas realizam quando estão inicializando. Com isto tem-se uma maior precisão do tempo de execução do algoritmo em si.

O número de iterações de *benchmark* é decidido pela biblioteca de testes da linguagem, esta realiza iterações até o tempo total estiver a aproximadamente um segundo, logo é necessário dividir este tempo total pelo número de iterações.

O programa construído possui uma interface de linha de comando com os seguintes subcomandos:

* sort método valores... : Ordena o vetor passado como argumento e retorna-o ordenado.
* bench método: Executa a suíte de benchmark do método especificado, a saída é em formato CSV.
* stepBench método: Executa a suíte de benchmark de dado método porém espera a máquina ficar abaixo da temperatura especificada para iniciar o próximo passo do teste. A temperatura é obtida apenas no Linux utilizando um arquivo existente na interface de sistema representada pela pasta /sys.
* autoBench: Realiza a mesma atividade que o stepBench, porém para todos os métodos e 3 vezes. Esta foi uma automação construída para reduzir possíveis falhas humanas durante um longo teste e foi o comando utilizado para os testes apresentados na seção de resultados.

O programa que constrói os vetores aleatórios é um programa que constrói o código Go de um programa e envia o texto para a saída padrão, a qual é redirecionada para um arquivo utilizando o operador “>”, por exemplo.

O compilador Go utilizado nos testes é a versão 1.13.1 executando em um notebook Acer Aspire 3 A315-51-51SL equipado com um processador i5 7200U, 6GB de memória RAM DDR4, 1TB de HD executando o sistema operacional Arch Linux sob o kernel Linux 5.3.1-arch1-1-ARCH.



Figura : Notebook realizando o teste.  
**Fonte:** Autor

A imagem acima demonstra o ambiente de teste. A máquina está com a bateria carregada e na fonte. Para ventilar melhor foi-se usado um ventilador comum. No notebook o programa está executando as operações uma a uma e imprimindo na tela, enquanto salva em arquivo os resultados das iterações em formato CSV. Mesmo com o ventilador, o próximo passo do teste só inicia quando a temperatura ficar abaixo de 43 graus Celsius. Os testes ocorreram no dia 17/10/2019 iniciando aproximadamente às 17:30 até aproximadamente 21:00.

Para a geração dos gráficos partindo dos dados em CSV será utilizada a linguagem R utilizando a função *qplot* da biblioteca *ggplot2.* Todos os programas baseados em tecnologias Web, ou que tenham alguma fama de serem pesados e não primordiais são fechados durante os testes a fim de que não interfiram nos resultados. Isto inclui Spotify, Firefox e o Tixati.

Por natureza do Arch Linux, descarta-se a possibilidade de este forçar ou realizar alguma atualização em segundo plano durante os testes.

Cada método terá seu gráfico. Ao final será mostrado o gráfico com todos os algoritmos para fins de comparação, cada um com uma cor diferente. Caso seja encontrado algum caso onde existam ordens de grandeza de diferença de tempo entre os métodos dificultando a observação então serão construídos gráficos mostrando apenas estes casos mais difíceis de observar.

Os vetores de todos os testes serão iguais e serão limitados de zero a um milhão. Nenhum elemento de nenhum vetor estará fora deste intervalo.

Cada método será executado três vezes com vetores de tamanho múltiplo de 5000 partindo de 5000 até 200000, ou seja 40 testes por algoritmo. O gráfico por ser de pontos facilita a visualização de eventuais variações entre iterações e também ajuda a entender a interferência do coletor de lixo nos testes.

Partindo da premissa acima tem-se: 40 testes por algoritmo vezes 8 algoritmos vezes três iterações totalizando 960 linhas. Se contar o cabeçalho do arquivo CSV, que tem como objetivo orientar o programa que está lendo sobre os campos presentes, o número de linhas se torna 961.

Ao final de cada passo o programa aguarda a temperatura da máquina atingir determinado critério. A temperatura será obtida utilizando um arquivo especial do sistema Linux que é o /sys/class/termal\_zone0/temp. Este arquivo tem sempre a leitura mais recente pois é uma interface especial do kernel Linux. O número obtido por este arquivo é a temperatura atual multiplicada por 1000. Esta medição é equivalente a obtida pelo *sensors*, programa utilizado para medição de temperatura em sensores. A função que obtém temperatura do sistema partindo da premissa acima é a seguinte, presente no arquivo main.go.

O teste realizado e apresentado nos resultados foi executado da seguinte forma:

Figura : Código da função getTemperatura utilizada para obter a temperatura atual da máquina  
**Fonte:** Autor

func getTemperatura() int {

strtemp, err := ioutil.ReadFile("/sys/class/thermal/thermal\_zone0/temp")

if err != nil {

panic(err)

}

strtemp = strtemp[0:len(strtemp) - 1] // Tira o \n no final da string, evitando o erro ao obter em número

temp, err := strconv.Atoi(string(strtemp))

if err != nil {

panic(err)

}

return temp/1000

}

* Primeiro foi-se compilado o programa que randomiza os vetores em um arquivo de código Go. Este arquivo foi guardado em um lugar onde o programa principal possa achar no processo de compilação.
* A execução dos testes foi feita usando o seguinte comando:

./gosort autoBench | tee –a dados.txt

* Os dados obtidos nos testes serão então salvos no arquivo dados.txt e exibidos na tela.

Com o arquivo dados.txt gerado após os testes, foi-se filtrado os resultados que são relevantes ao teste, como resultados filtrados por um conjunto de métodos ou de apenas um, em um outro arquivo e este arquivo então é passado como parâmetro para o script R que confecciona o gráfico. Este script aceita como parâmetro um método que é filtrado, para mais métodos é necessário fazer a filtragem de outro modo, por exemplo, usando o comando grep.

Para executar o script o comando é:

Rscript criarGráfico.r arquivo método

Ao executar o script, o gráfico fica presente em formato de imagem SVG com o nome resultado.

# RESULTADOS E DISCUSSÃO

Com os testes realizados chegou-se nos seguintes resultados:

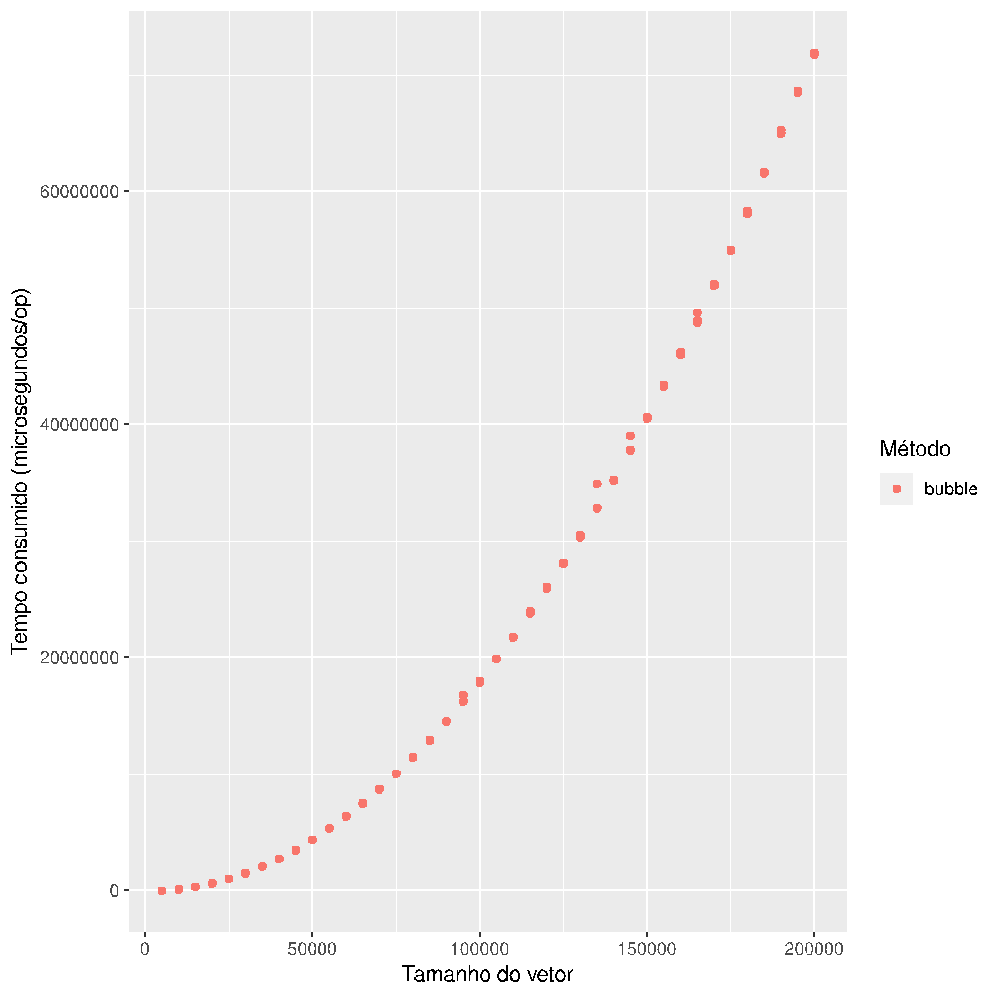


Figura : Resultado do *benchmark* utilizando o algoritmo *bubble sort.***Fonte:** Autor

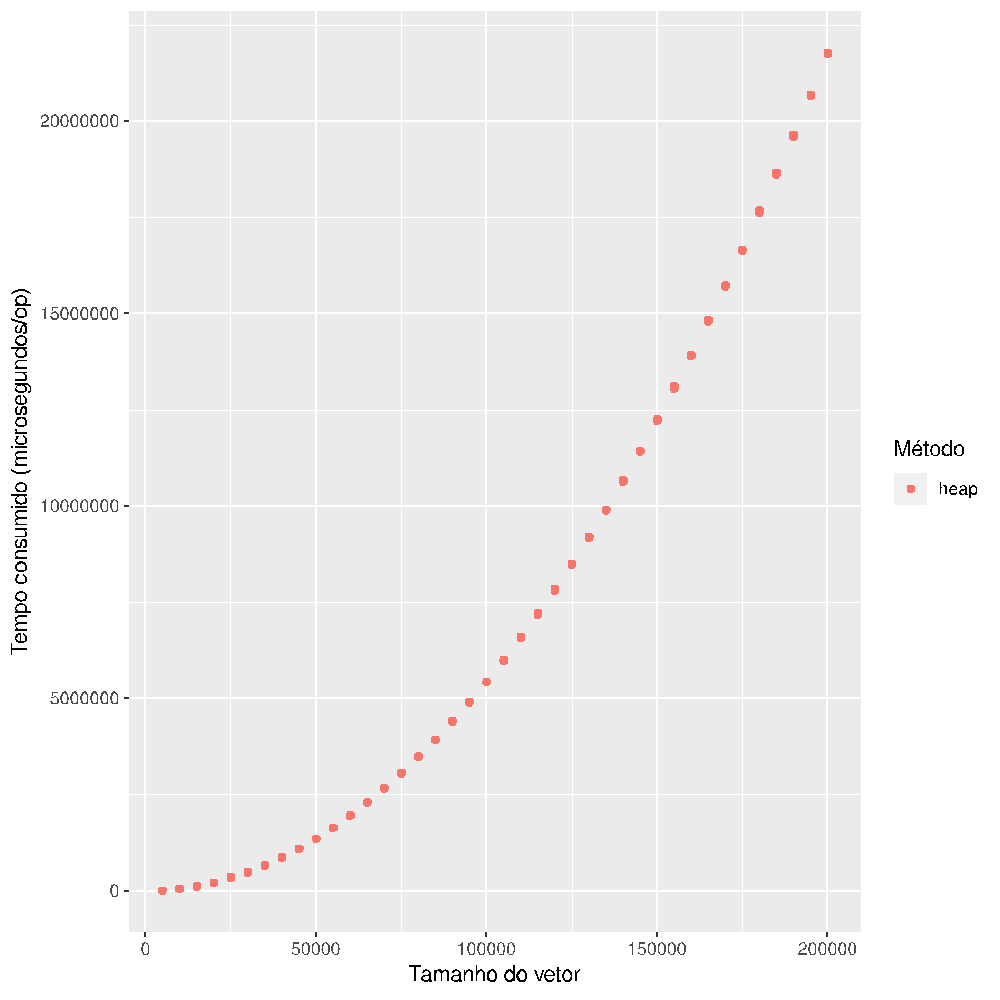


Figura : Resultado do benchmark para o algoritmo *heap sort.***Fonte:** Autor

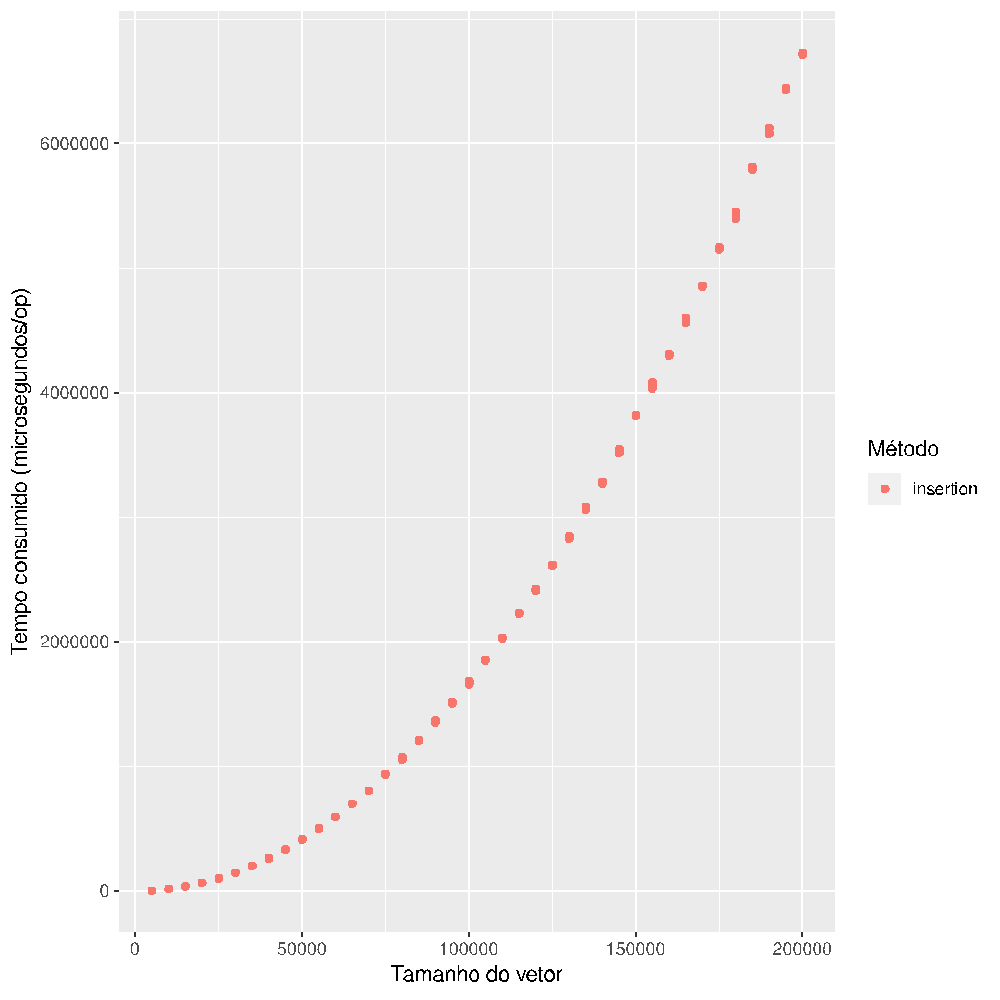


Figura : Resultado do benchmark para o algoritmo *insertion sort.***Fonte:** Autor

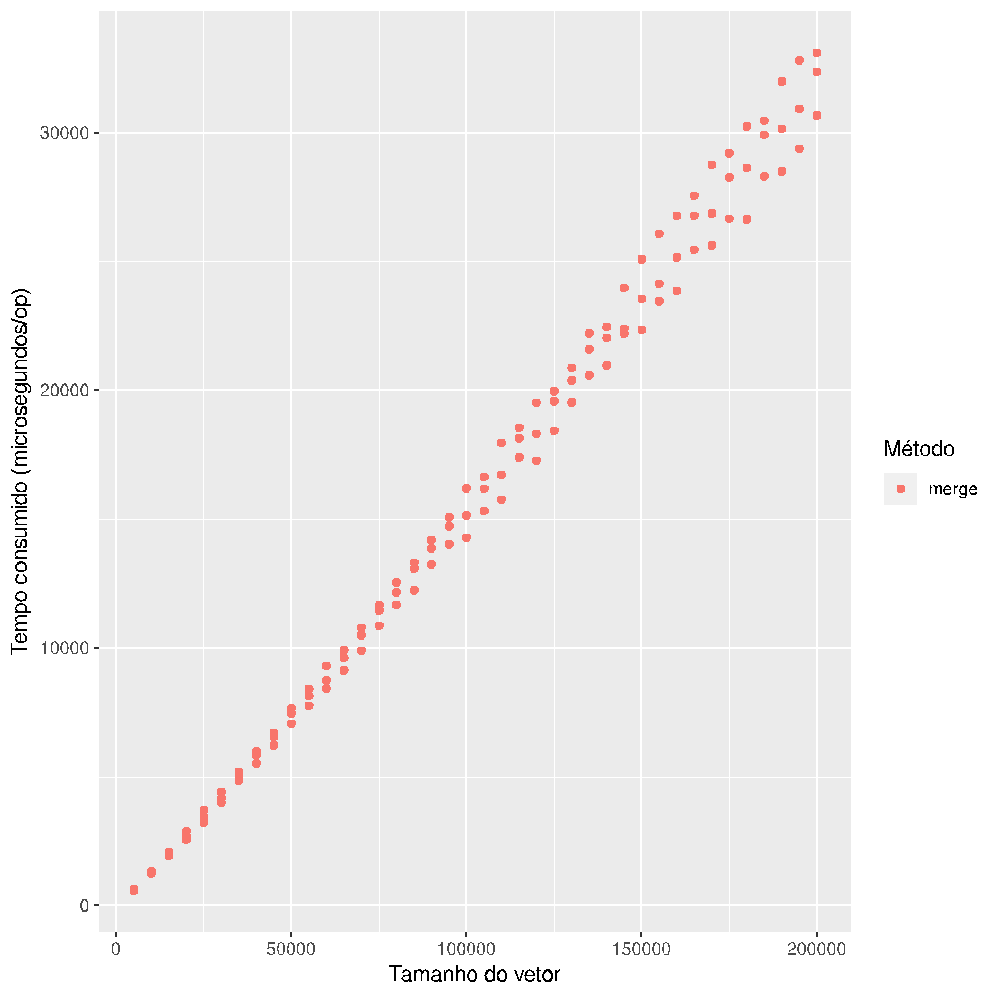


Figura : Resultado do benchmark para o algoritmo *merge sort.***Fonte:** Autor

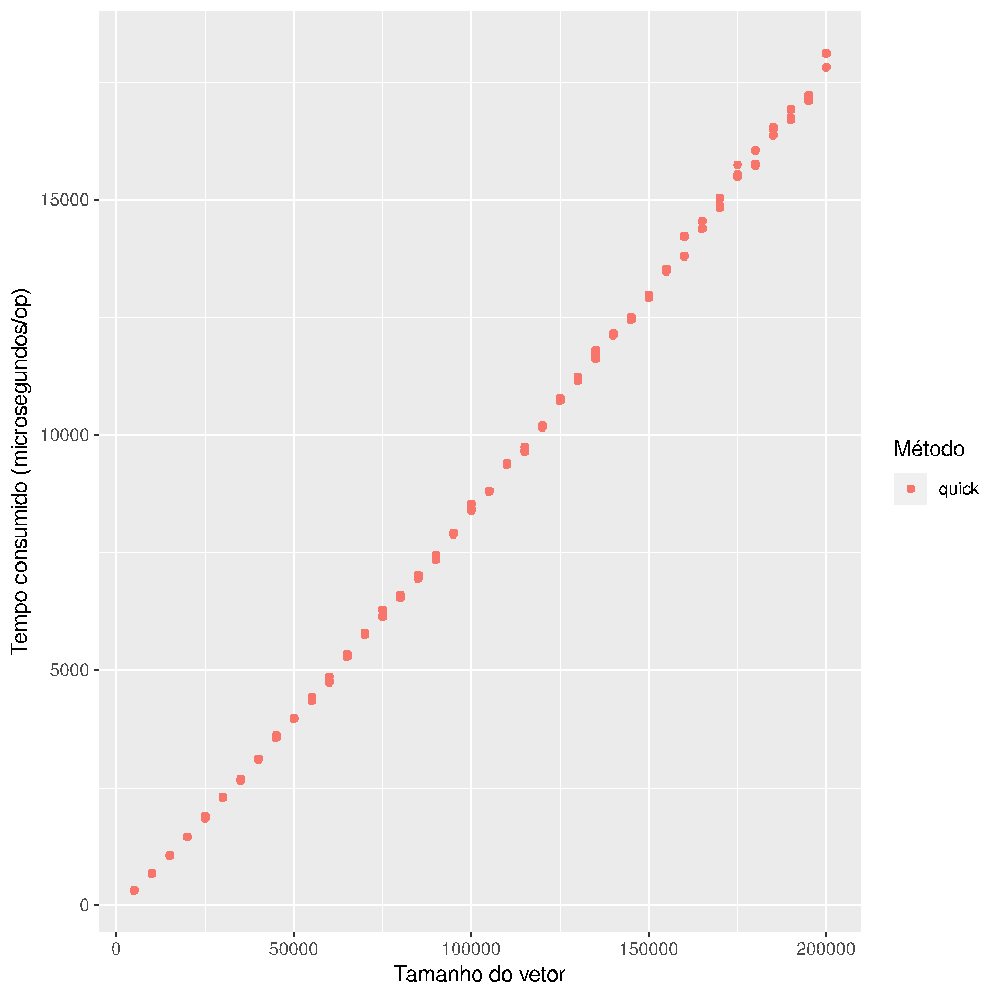


Figura Resultado do benchmark para o algoritmo *quick sort.*

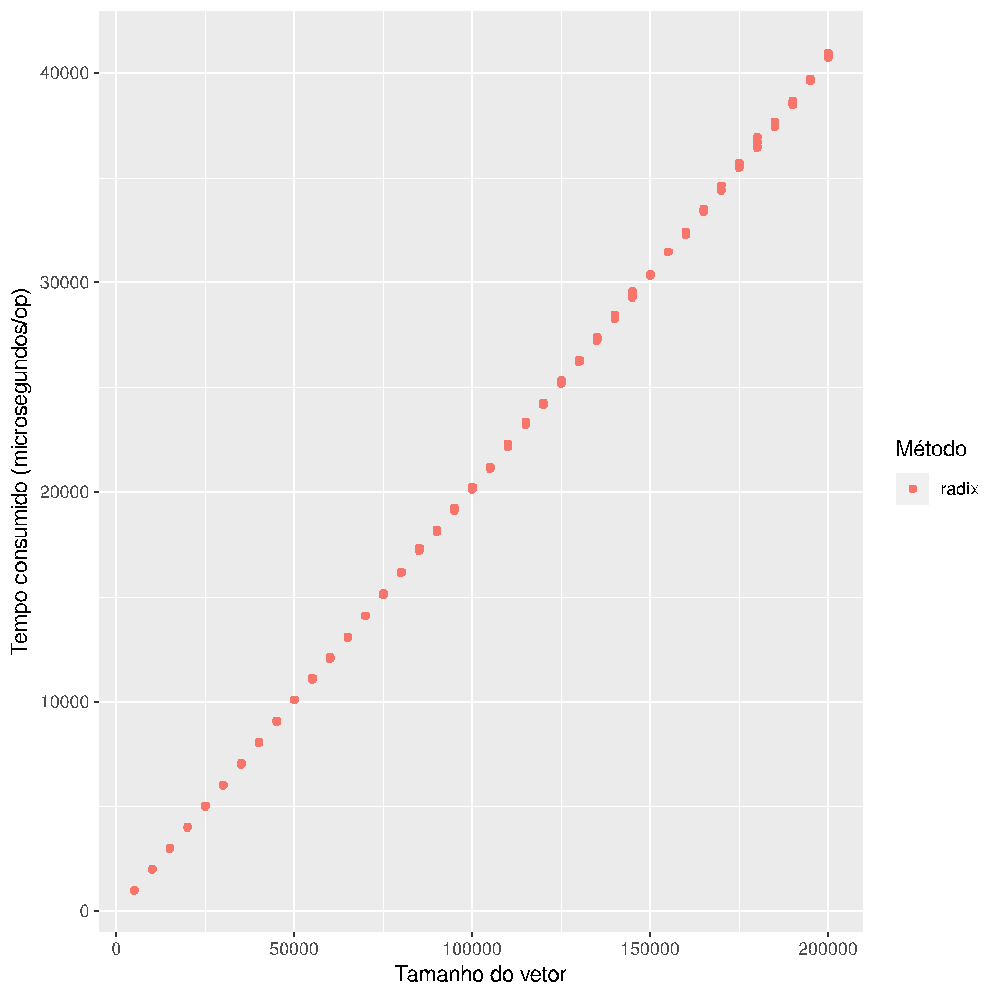


Figura Resultado do benchmark para o algoritmo *radix sort.***Fonte:** Autor

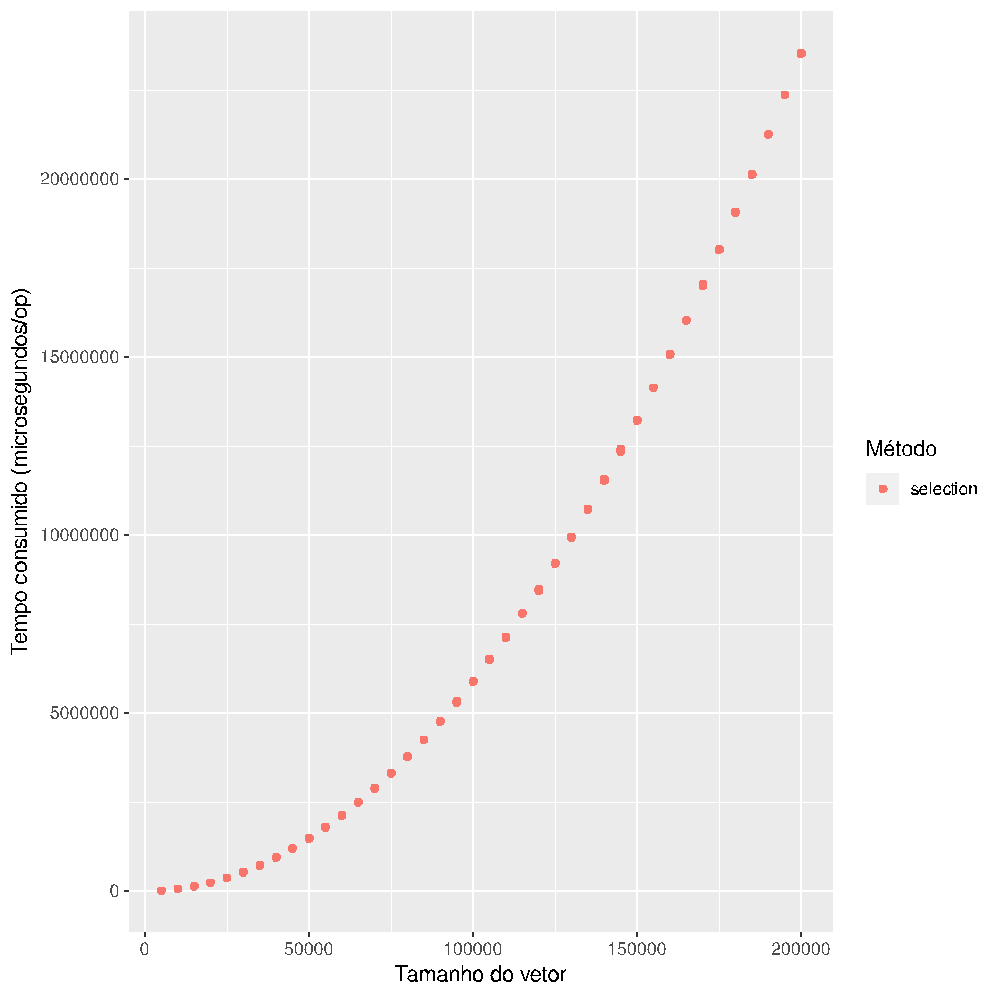


Figura Resultado do benchmark para o algoritmo *selection sort.***Fonte:** Autor

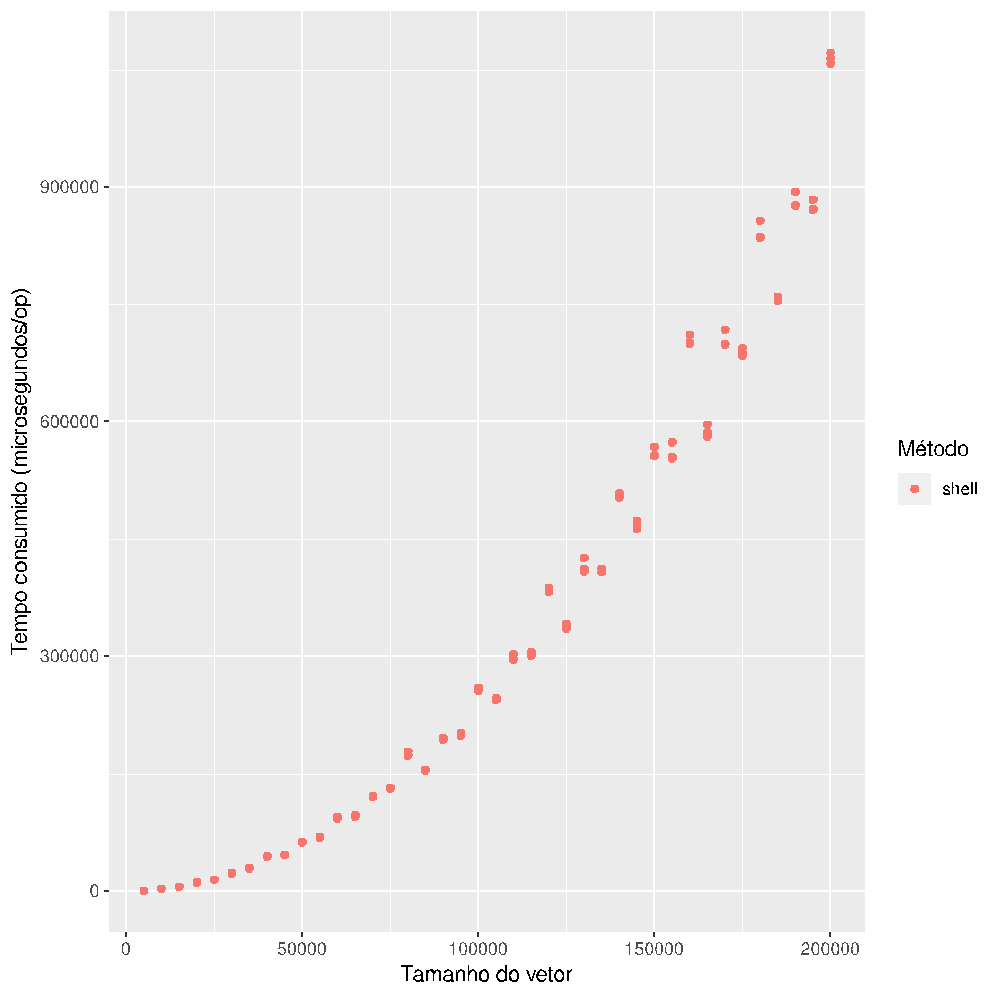


Figura Resultado do benchmark para o algoritmo *selection sort.***Fonte:** Autor

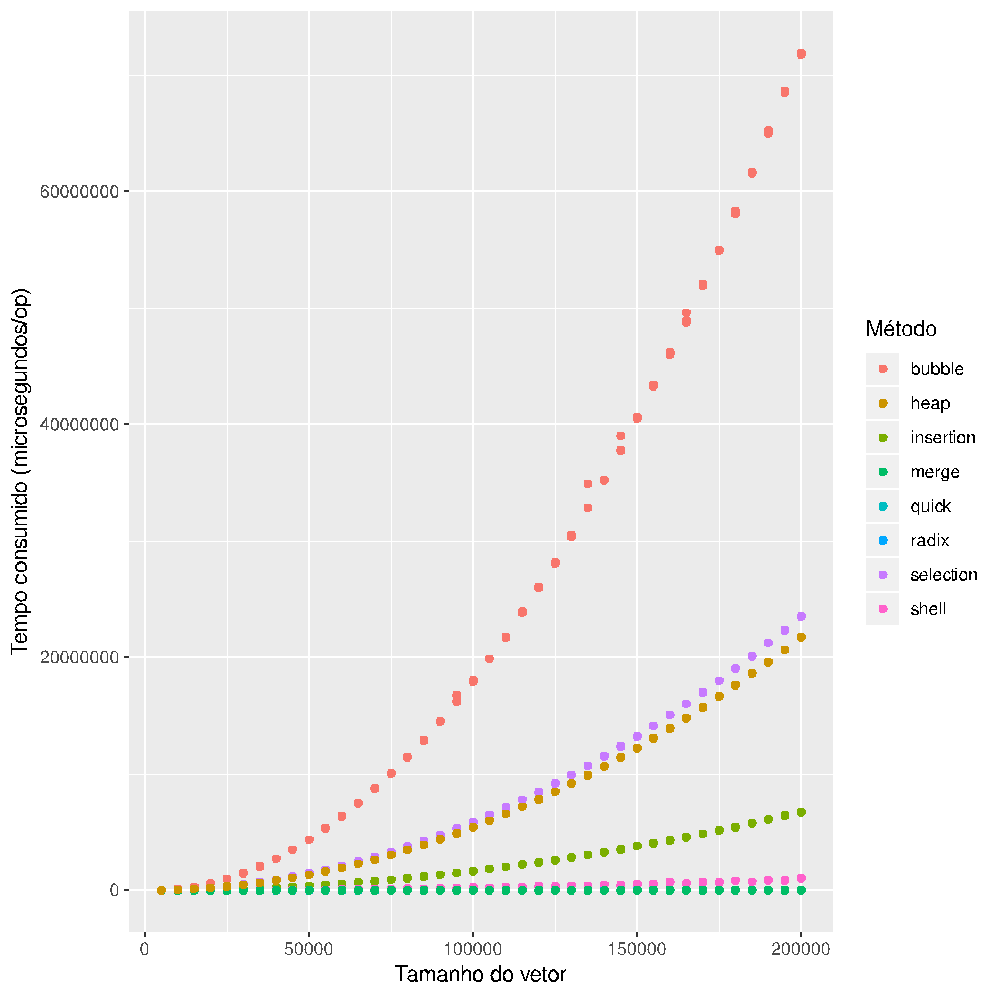


Figura : Resultado unificado do *benchmark* de todos os algoritmos testados.  
**Fonte:** Autor

Nestes gráficos demonstrados acima percebe-se que três dos quatro algoritmos tem resultados muito próximos e muito baixos em comparação com o *bubble sort*.

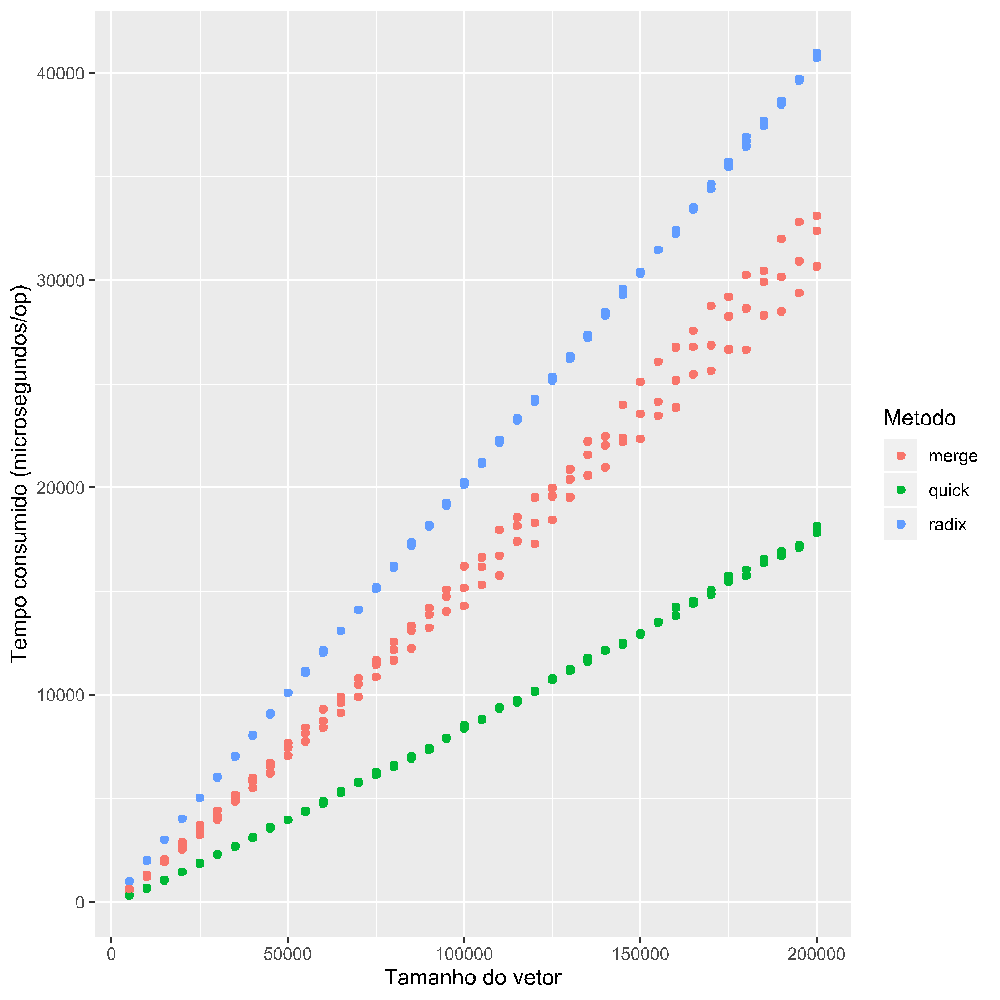


Figura : Resultado do *benchmark* dos 3 algoritmos mais eficientes.  
**Fonte:** Autor

Ao limitar o gráfico apenas nos três algoritmos mais eficientes percebe-se claramente que o algoritmo mais eficiente é o *quick sort*, seguido do *merge sort* e do *radix sort*.

Percebe-se também o quanto um algoritmo pode mudar drasticamente o tempo de execução de uma tarefa. Em um total de aproximadamente 40 milissegundos o *quick sort* conseguiu ordenar um vetor de 200000 valores.

**CONSIDERAÇÕES FINAIS**

O objetivo deste trabalho é comparar a eficiência dos principais métodos de ordenação de dados aplicados a vetores de números inteiros implementando-os e comparando o tempo utilizado para ordenar vetores de diferentes tamanhos com números aleatórios. Os 8 algoritmos testados foram o *bubble*, *heap*, *insertion*, *merge*, *quick*, *radix*, *selection* e *shell sort* e foram implementados na linguagem Go.

Os resultados confirmam que este objetivo foi atendido, pois é possível ver claramente qual algoritmo obteve melhor performance e qual não obteve uma performance tão boa. Este trabalho utilizou os algoritmos em sua forma mais simples, sendo executados em apenas um fluxo de execução, O código criado para este artigo pode ser adaptado para ser utilizado como um framework para testes de algoritmos de ordenação, o que é um trabalho futuro sugerido.

**REFERÊNCIAS**

CHAND, S. et al.  Upgraded Selection Sort. **International Journal on Computer Science and Engineering (IJCSE)***,*Ghaziabad, U.P., India, 3, 4, 1756 p., 4 abr. 2011. Disponível em: < http://www.enggjournals.com/ijcse/doc/IJCSE11-03-04-115.pdf>. Acesso em 18 out. 2019.

DUTTA, P. S.  An Approach to Improve the Performance of Insertion Sort Algorithm. **International Journal of Computer Science & Engineering Technology***,*West Bengal, India, 4, 5, 630p, 5 mai. 2013. Disponível em: < http://www.ijcset.com/docs/IJCSET13-04-05-068.pdf>. Acesso em: 18 out. 2019.

FRANCIS, R. S.; MATHIESON, I. D. A benchmark parallel sort for shared memory multiprocessors. **IEEE Transactions on Computers**, v. 37, n. 12, p. 1619–1626, dez. 1988.

GARCIA, A. M. et al. Analisando o Desempenho do Mergesort In-Place Paralelo em Duas Arquiteturas Multi-core. 2014.

GOLANG. **The Go Programming Language Specification - The Go Programming Language**. Disponível em: <https://golang.org/ref/spec>. Acesso em: 18 out. 2019.

HOARE, C. A. R. Quicksort. **The Computer Journal**, v. 5, n. 1, p. 10–16, 1 jan. 1962.

HORSMALAHTI, P.  Comparison of Bucket Sort and RADIX Sort. **ArXiv**, 10 p., 18 jun. 2012.  Disponível em: < https://arxiv.org/pdf/1206.3511.pdf>. Acesso em: 18 out. 2019.

JADOON, S.; SOLEHRIA, S. F.; QAYUM, M. Optimized Selection Sort Algorithm is faster than Insertion Sort Algorithm: a Comparative Study. **Internacional Journal of Electrical & Computer Sciences IJECS-IJENS***,*Haripur, 11, 02, 5 p., abr. 2012. Disponível em: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.654.4716&rep=rep1&type=pdf>. Acesso em 18 out. 2019.

MIN, W. **Analysis on Bubble Sort Algorithm Optimization**. 2010 International Forum on Information Technology and Applications. **Anais**... In: 2010 INTERNATIONAL FORUM ON INFORMATION TECHNOLOGY AND APPLICATIONS. jul. 2010.

QIN, S. Merge Sort Algorithm. **Dept. Of Computer Science – Florida Institute of Technology**, Melbourne. Disponível em: <https://pdfs.semanticscholar.org/6804/987ab63d1879aa55ba68224dced142ce8774.pdf>. Acesso em: 18 out. 2019.

SCHAFFER, R.; SEDGEWICK, R. The Analysis of Heapsort. **Journal of Algorithms**, v. 15, n. 1, p. 76–100, 1 jul. 1993.

SEDGEWICK, R. **Analysis of Shellsort and related algorithms**. (J. Diaz, M. Serna, Eds.)Algorithms — ESA ’96. **Anais**...: Lecture Notes in Computer Science.Springer Berlin Heidelberg, 1996.

# ANEXOS

Anexo 1 – Código desenvolvido e utilizado no trabalho.

go.mod: Este arquivo é utilizado pelo compilador go para identificar o nome do pacote raiz do programa. Isto permite realizar importações de membros de um mesmo pacote sem que este tenha que ficar em uma pasta específica. Esta funcionalidade foi introduzida na versão 1.11 da linguagem.

module gosort

go 1.12

main.go: Este arquivo é o programa principal. Este que será utilizado como ponto de entrada do programa importando os outros componentes do programa. Rotinas de teste não precisam necessariamente declarar um ponto de entrada.

package main

import (

    "fmt"

    "gosort/benchmark"

    "gosort/bubble"

    "gosort/heap"

    "gosort/insertion"

    "gosort/merge"

    "gosort/quick"

    "gosort/radix"

    "gosort/selection"

    "gosort/shell"

    "gosort/vector"

    "io/ioutil"

    "os"

    "strconv"

    "time"

)

var comandos = map[string]func([]int) []int{

    "bubble":    bubble.Bubble,

    "heap":      heap.Heap,

    "insertion": insert.Insert,

    "radix":     radix.Radix,

    "selection": selection.Selection,

    "shell":     shell.Shell,

    "quick":     quick.Quick,

    "merge":     merge.Merge,

}

var operacoes = map[string]func(){

    "sort":      cmdSort,

    "bench":     cmdBench,

    "stepBench": cmdStepBench,

    "autoBench": cmdAutoBench,

}

func main() {

    if len(os.Args) < 2 {

        fmt.Printf("Comandos disponiveis: ")

        for k := range operacoes {

            fmt.Printf(" %s", k)

        }

        fmt.Printf("\n")

        panic("Argumentos insuficientes")

    }

    operacoes[os.Args[1]]()

}

func cmdSort() {

    if len(os.Args) < 3 {

        fmt.Printf("Comandos disponiveis: ")

        for k := range comandos {

            fmt.Printf(" %s", k)

        }

        fmt.Printf("\n")

        panic("Argumentos insuficientes")

    }

    cmd := comandos[os.Args[2]]

    var err error

    numeros := make([]int, len(os.Args)-3)

    for i := 3; i < len(os.Args); i++ {

        numeros[i-3], err = strconv.Atoi(os.Args[i])

        if err != nil {

            panic(err)

        }

    }

    fmt.Printf("%v\n", cmd(numeros))

}

var isStepBench = false

func cmdStepBench() {

    isStepBench = true

    cmdBench()

}

func getTemperatura() int {

    strtemp, err := ioutil.ReadFile("/sys/class/thermal/thermal\_zone0/temp")

    if err != nil {

        panic(err)

    }

    strtemp = strtemp[0 : len(strtemp)-1] // Tira o \n no final da string, evitando o erro ao obter em número

    temp, err := strconv.Atoi(string(strtemp))

    if err != nil {

        panic(err)

    }

    return temp / 1000

}

func cmdAutoBench() {

    isStepBench = true

    printCSVHeader()

    for i := 0; i < 3; i++ {

        for k, v := range comandos {

            handleBenchMethod(k, v)

        }

    }

}

const esperarTemperatura = 43 // Esperar até a máquina chegar a x graus para testar o próximo vetor

func cmdBench() {

    if len(os.Args) < 3 {

        fmt.Printf("Comandos disponiveis: ")

        for k := range comandos {

            fmt.Printf(" %s", k)

        }

        fmt.Printf("\n")

        panic("Argumentos insuficientes")

    }

    f := comandos[os.Args[2]]

    printCSVHeader()

    handleBenchMethod(os.Args[2], f)

}

func printCSVHeader() {

    fmt.Println("metodo,tamanhovetor,tempo")

}

func handleBenchMethod(algorithm string, f func([]int) []int) {

    ticker := time.Tick(time.Second)

    for k, v := range vector.Vector {

        res := benchmark.ResultifyBenchmark(benchmark.BenchmarkifyCase(f, v))

        fmt.Printf("%s,%d,%.2f\n", algorithm, k, float64(res.T.Microseconds())/float64(res.N))

        for isStepBench && getTemperatura() > esperarTemperatura { // Espera até a temperatura ficar menor que esperarTemperatura, automações...

            <-ticker

        }

    }

}

criarGrafico.r: Programa em R que carrega o arquivo CSV construído pelos testes e cria um gráfico e salva em um arquivo chamado resultado.

args = commandArgs(trailingOnly = TRUE)

if(length(args) < 1) {

    stop("Nome do arquivo para entrada não encontrado!")

}

require('ggplot2')

# png('resultado.png')

svg('resultado.svg')

f = read.csv(args[1]) # Le o arquivo passado como parametro

f <- within(f, {

    tempo <- as.numeric(as.character(tempo)) # Converte o tempo de string para numerico

    tamanhovetor <- as.numeric(as.character(tamanhovetor)) # Converte tamanho do vetor de string para numérico

})

if(length(args) == 2) {

    f = f[grep(args[2], f$metodo), ]

}

options(scipen=max(f$tempo)) # Tira notação científica

Metodo = f$metodo

qplot(x=f$tamanhovetor, y=f$tempo, colour = Metodo, xlab = "Tamanho do vetor", ylab = "Tempo consumido (microsegundos/op)")

dev.off()

benchmark/benchmark.go: Este arquivo contém funções utilitárias para construir um objeto *benchmark* dada uma função que tenha um formato coincidente com a dos métodos de ordenação e para obter os resultados da execução de um *benchmark*.

package benchmark

import (

    "gosort/vector"

    "testing"

)

func BenchmarkMethod(f func([]int) []int) map[int]testing.BenchmarkResult {

    results := map[int]testing.BenchmarkResult{}

    for k, v := range vector.Vector {

        benchfunc := BenchmarkifyCase(f, v)

        results[k] = ResultifyBenchmark(benchfunc)

    }

    return results

}

func ResultifyBenchmark(f func(b \*testing.B)) testing.BenchmarkResult {

    return testing.Benchmark(f)

}

func BenchmarkifyCase(f func([]int) []int, vet []int) func(b \*testing.B) {

    ord := make([]int, len(vet))

    return func(b \*testing.B) {

        for n := 0; n < b.N; n++ {

            copy(ord, vet) // Copia o vetor salvo lá para outro lugar, já que passamos ele como referencia

            f(ord)

        }

    }

}

bubble/bubble\_test.go: Este arquivo contém uma função em formato de teste que passa a função de ordenação do *bubble sort,* onde são aplicados todos os casos de teste de forma generalizada.

package bubble

import (

    "gosort/test\_util"

    "testing"

)

func TestBubbleSort(t \*testing.T) {

    util.CheckOrdenacao(Bubble)(t)

}

bubble/bubble.go: Implementação do *bubble sort*

package bubble

// Bubble sort

func Bubble(valores []int) []int {

    tamanho := len(valores)

    isAlterado := true

    for isAlterado {

        isAlterado = false

        for i := 0; i < tamanho-1; i++ {

            if valores[i] > valores[i+1] {

                dummy := valores[i]

                valores[i] = valores[i+1]

                valores[i+1] = dummy

                isAlterado = true

            }

        }

    }

    return valores

}

heap/heap\_test.go: Este arquivo contém uma função em formato de teste que passa a função de ordenação do *heap sort,* onde são aplicados todos os casos de teste de forma generalizada.

package heap

import (

    "gosort/test\_util"

    "testing"

)

func TestHeapSort(t \*testing.T) {

    util.CheckOrdenacao(Heap)(t)

}

heap/heap.go: Implementação do *heap sort.*

package heap

// Heap Sort

func Heap(valores []int) []int {

    for max := len(valores) - 1; max > 0; max-- {

        imax := max

        i := 0

        for ; i < max; i++ {

            if valores[i] > valores[imax] {

                imax = i

            }

        }

        dummy := valores[max]

        valores[max] = valores[imax]

        valores[imax] = dummy

    }

    return valores

}

insertion/insert\_test.go: Este arquivo contém uma função em formato de teste que passa a função de ordenação do *insertion sort,* onde são aplicados todos os casos de teste de forma generalizada.

package insert

import (

    "gosort/test\_util"

    "testing"

)

func TestInsertSort(t \*testing.T) {

    util.CheckOrdenacao(Insert)(t)

}

insertion/insert.go: Implementação do insertion sort.

package insert

// Insert sort

func Insert(vetor []int) []int {

    tamanho := len(vetor)

    for i := 1; i < tamanho; i++ {

        dummy := vetor[i] // Valor da reserva

        j := i - 1

        for ; j >= 0 && vetor[j] > dummy; j-- {

            vetor[j+1] = vetor[j]

        }

        vetor[j+1] = dummy

    }

    return vetor

}

merge/merge\_test.go: Este arquivo contém uma função em formato de teste que passa a função de ordenação do *merge sort,* onde são aplicados todos os casos de teste de forma generalizada.

package merge

import (

    "gosort/test\_util"

    "testing"

)

func TestMergeSort(t \*testing.T) {

    util.CheckOrdenacao(Merge)(t)

}

merge/merge.go: Implementação do merge sort

package merge

// Merge sort

func Merge(valores []int) []int {

    tamanho := len(valores)

    if tamanho <= 1 {

        return valores

    }

    metade := tamanho / 2

    va := valores[0:metade]

    vb := valores[metade:tamanho]

    Merge(va)

    Merge(vb)

    ia := 0

    ib := 0

    tmp := make([]int, tamanho)

    for i := 0; i < tamanho; i++ {

        if ib == len(vb) {

            tmp[i] = va[ia]

            ia++

            continue

        }

        if ia < len(va) && va[ia] <= vb[ib] {

            tmp[i] = va[ia]

            ia++

        } else if ib < len(vb) {

            tmp[i] = vb[ib]

            ib++

        }

    }

    for i := 0; i < tamanho; i++ {

        valores[i] = tmp[i]

    }

    tmp = nil

    return valores

}

quick/quick\_test.go: Este arquivo contém uma função em formato de teste que passa a função de ordenação do *quick sort,* onde são aplicados todos os casos de teste de forma generalizada.

package quick

import (

    "gosort/test\_util"

    "testing"

)

func TestQuickSort(t \*testing.T) {

    util.CheckOrdenacao(Quick)(t)

}

quick/quick.go: Implementação do quick sort

package quick

// Quick sort

func Quick(valores []int) []int {

    qsort(valores, 0, len(valores)-1)

    return valores

}

func qsort(valores []int, low, high int) {

    if low < high {

        pi := partition(valores, low, high)

        qsort(valores, low, pi-1)

        qsort(valores, pi+1, high)

    }

}

func partition(valores []int, low, high int) int {

    pivot := valores[high]

    i := (low - 1)

    for j := low; j <= high-1; j++ {

        if valores[j] < pivot {

            i++

            swp := valores[i]

            valores[i] = valores[j]

            valores[j] = swp

        }

    }

    swp := valores[i+1]

    valores[i+1] = valores[high]

    valores[high] = swp

    return i + 1

}

radix/radix\_test.go: Este arquivo contém uma função em formato de teste que passa a função de ordenação do *radix sort,* onde são aplicados todos os casos de teste de forma generalizada.

package radix

import (

    "gosort/test\_util"

    "testing"

)

func TestRadixSort(t \*testing.T) {

    util.CheckOrdenacao(Radix)(t)

}

radix/radix.go: Implementação do radix sort

package radix

// Radix Sort

func Radix(valores []int) []int {

    dummyl := make([]int, len(valores))

    maior := valores[0]

    exp := 1

    for i := 0; i < len(valores); i++ {

        if valores[i] > maior {

            maior = valores[i]

        }

    }

    for maior/exp > 0 {

        buck := make([]int, 10)

        for i := 0; i < len(valores); i++ {

            buck[(valores[i]/exp)%10]++

        }

        for i := 1; i < 10; i++ {

            buck[i] += buck[i-1]

        }

        for i := len(valores) - 1; i >= 0; i-- {

            buck[(valores[i]/exp)%10]--

            dummyl[buck[(valores[i]/exp)%10]] = valores[i]

        }

        for i := 0; i < len(valores); i++ {

            valores[i] = dummyl[i]

        }

        exp \*= 10

    }

    return valores

}

rand/rand.go: Implementação de gerador de números aleatórios usado nos testes unitários. Os vetores do benchmark são gerados antes de compilar o programa principal.

package generator

import (

    "math/rand"

)

func Generate(n int) []int {

    ret := make([]int, n)

    for i := 0; i < n; i++ {

        ret[i] = rand.Int()

    }

    return ret

}

selection/selection\_test.go: Este arquivo contém uma função em formato de teste que passa a função de ordenação do *selection sort,* onde são aplicados todos os casos de teste de forma generalizada.

package selection

import (

    "gosort/test\_util"

    "testing"

)

func TestSelectionSort(t \*testing.T) {

    util.CheckOrdenacao(Selection)(t)

}

selection/selection.go: Implementação do selection sort.

package selection

// Selection sort

func Selection(valores []int) []int {

    for i := 0; i < len(valores)-1; i++ {

        minimo := i

        for j := i; j < len(valores); j++ {

            if valores[minimo] > valores[j] {

                minimo = j

            }

        }

        if i != minimo {

            swp := valores[i]

            valores[i] = valores[minimo]

            valores[minimo] = swp

        }

    }

    return valores

}

ARQUIVO: shell/shell\_test.go: Este arquivo contém uma função em formato de teste que passa a função de ordenação do *shell sort,* onde são aplicados todos os casos de teste de forma generalizada.

package shell

import (

    "gosort/test\_util"

    "testing"

)

func TestShellSort(t \*testing.T) {

    util.CheckOrdenacao(Shell)(t)

}

ARQUIVO: shell/shell.go: Implementação do shell sort

package shell

import (

    "gosort/insertion"

)

func Shell(valores []int) []int {

    hn := roundupDivision(len(valores))

    for ; hn > 1; hn = roundupDivision(hn) {

        for i := 0; i < len(valores)-hn; i++ {

            if valores[i] > valores[i+hn] {

                swp := valores[i]

                valores[i] = valores[i+hn]

                valores[i+hn] = swp

            }

        }

    }

    return insert.Insert(valores)

}

func roundupDivision(n int) int {

    ret := n / 2

    if (n % 2) != 0 {

        ret++

    }

    return ret

}

test\_util/util\_test.go: Testes da função que testa se o vetor está ordenado.

package util

import (

    "testing"

)

func TestIsOrdenado(t \*testing.T) {

    truthCases := [][]int{

        {2, 3, 4, 5, 6},

        {1, 66, 777, 832},

        {1, 2, 3, 4},

        {1},

    }

    for i := range truthCases {

        if !IsOrdenado(truthCases[i]) {

            t.Fail()

        }

    }

    falsyCases := [][]int{

        {5, 4, 3, 2, 1},

        {99, 0},

        {9, 1},

    }

    for i := range falsyCases {

        if IsOrdenado(falsyCases[i]) {

            t.Fail()

        }

    }

}

test\_util/util.go: Definição da função que checa se um vetor está ordenado, da função que constrói uma função de teste dada a função do algoritmo e dos casos de teste dos algoritmos de ordenação.

package util

import (

    "gosort/rand"

    "testing"

)

var cases = map[string][]int{

    "123":          {1, 2, 3},

    "321":          {3, 2, 1},

    "alternado":    {1, 3, 5, 7, 9, 2, 4, 6, 8},

    "aleatorio10":  generator.Generate(10),

    "aleatorio100": generator.Generate(100),

}

// IsOrdenado o vetor está ordenado?

func IsOrdenado(vet []int) bool {

    for i := 0; i < len(vet)-1; i++ {

        if vet[i] > vet[i+1] {

            return false

        }

    }

    return true

}

// CheckOrdenacao retorna test case checando ordenacao

func CheckOrdenacao(f func([]int) []int) func(\*testing.T) {

    return func(t \*testing.T) {

        for k, v := range cases {

            this := make([]int, len(v))

            copy(this, v)

            t.Run(k, func(t \*testing.T) {

                isOrdenado := IsOrdenado(f(this))

                if !isOrdenado {

                    t.Errorf("%#v nao foi ordenado corretamente\n", this)

                }

            })

        }

    }

}

vector/generator/main.go: Programa que gera vetores aleatórios e constrói um pacote Go com estes vetores para ser compilado em conjunto com o programa principal.

package main

import (

    "html/template"

    "math/rand"

    "os"

)

func Generate(n int) []int {

    ret := make([]int, n)

    for i := 0; i < n; i++ {

        ret[i] = rand.Intn(1000000)

    }

    return ret

}

var tpl = `package vector

var Vector = map[int][]int{

    {{ range $k, $v := . }}

        {{ $k }}: []int{ {{ range $num := $v }} {{ $num }}, {{ end }} },

    {{end}}

}

`

func main() {

    t, err := template.New("vetores").Parse(tpl)

    if err != nil {

        panic(err)

    }

    if err != nil {

        panic(err)

    }

    cenarios := []int{}

    for i := 5000; i <= 200000; i += 5000 {

        cenarios = append(cenarios, i)

    }

    vetores := gerarVetores(cenarios)

    err = t.Execute(os.Stdout, vetores)

    if err != nil {

        panic(err)

    }

}

func gerarVetores(cenarios []int) map[int][]int {

    ret := map[int][]int{}

    for \_, cenario := range cenarios {

        ret[cenario] = Generate(cenario)

    }

    return ret

}

vector/aqui.go: Neste arquivo consta a saída que o programa acima retorna. Uma forma de obter este arquivo presumindo-se que o terminal está apontado para a raiz do projeto, ou seja, a mesma pasta onde reside o arquivo main.c, é executar o seguinte comando:

go run vector/generator/main.go > vector/aqui.go

Este arquivo não foi incluído neste trabalho por questões de tamanho. O arquivo gerado possui por volta de 80 MB.

dadosObtidos.txt: Este arquivo contém os dados obtidos no teste.

metodo,tamanhovetor,tempo

shell,90000,196093.67

shell,5000,751.74

shell,10000,2931.76

shell,15000,5702.53

shell,40000,44633.46

shell,50000,62506.00

shell,55000,69336.75

shell,80000,178350.67

shell,95000,201865.60

shell,35000,29424.89

shell,115000,305269.75

shell,130000,425978.33

shell,135000,412217.00

shell,165000,596231.00

shell,30000,23123.45

shell,60000,94961.67

shell,120000,387442.33

shell,145000,473314.67

shell,155000,573625.50

shell,100000,259941.00

shell,110000,302783.75

shell,170000,717707.00

shell,180000,856952.00

shell,190000,893716.00

shell,25000,14652.05

shell,150000,567775.00

shell,160000,711245.50

shell,20000,11049.31

shell,195000,883891.00

shell,70000,121359.22

shell,105000,246611.80

shell,140000,508577.00

shell,200000,1071686.00

shell,45000,46717.28

shell,65000,96737.75

shell,75000,132256.50

shell,85000,155009.57

shell,125000,341587.00

shell,175000,694101.50

shell,185000,760082.00

quick,70000,5791.05

quick,105000,8799.26

quick,140000,12153.73

quick,200000,18099.04

quick,45000,3564.38

quick,65000,5298.23

quick,75000,6290.13

quick,85000,6961.85

quick,125000,10790.75

quick,175000,15540.42

quick,185000,16504.88

quick,90000,7384.33

quick,5000,318.62

quick,10000,675.52

quick,15000,1064.79

quick,40000,3109.90

quick,50000,3972.67

quick,55000,4421.90

quick,80000,6604.93

quick,95000,7895.89

quick,35000,2679.24

quick,115000,9751.86

quick,130000,11234.27

quick,135000,11800.51

quick,165000,14394.09

quick,30000,2288.61

quick,60000,4861.77

quick,120000,10203.71

quick,145000,12511.43

quick,155000,13538.91

quick,100000,8419.98

quick,110000,9389.03

quick,170000,15037.19

quick,180000,15764.88

quick,190000,16735.26

quick,25000,1901.69

quick,150000,12975.66

quick,160000,14231.81

quick,20000,1456.78

quick,195000,17111.25

merge,145000,22201.75

merge,155000,23469.27

merge,30000,3996.99

merge,60000,8421.06

merge,120000,17280.86

merge,180000,26652.58

merge,190000,28506.95

merge,100000,14290.83

merge,110000,15759.07

merge,170000,25645.04

merge,25000,3245.97

merge,150000,22351.63

merge,160000,23866.12

merge,20000,2549.92

merge,195000,29388.40

merge,70000,9902.18

merge,105000,15320.27

merge,140000,20982.51

merge,85000,12246.80

merge,125000,18444.45

merge,175000,26666.53

merge,185000,28315.64

merge,200000,30682.78

merge,45000,6224.36

merge,65000,9140.82

merge,75000,10868.36

merge,40000,5518.17

merge,50000,7069.15

merge,55000,7767.17

merge,80000,11677.89

merge,90000,13250.87

merge,5000,589.54

merge,10000,1239.00

merge,15000,1934.66

merge,95000,14037.38

merge,135000,20589.72

merge,165000,25465.51

merge,35000,4863.96

merge,115000,17401.61

merge,130000,19543.05

bubble,35000,2058048.00

bubble,115000,23833845.00

bubble,130000,30383499.00

bubble,135000,32836769.00

bubble,165000,48954793.00

bubble,30000,1490590.00

bubble,60000,6392146.00

bubble,120000,25992026.00

bubble,145000,37833448.00

bubble,155000,43307818.00

bubble,100000,17912697.00

bubble,110000,21717792.00

bubble,170000,51908793.00

bubble,180000,58164516.00

bubble,190000,65000009.00

bubble,25000,1012197.00

bubble,150000,40527275.00

bubble,160000,46059555.00

bubble,20000,625203.50

bubble,195000,68634147.00

bubble,70000,8770139.00

bubble,105000,19887696.00

bubble,140000,35244201.00

bubble,175000,54993618.00

bubble,185000,61659462.00

bubble,200000,71783927.00

bubble,45000,3519342.00

bubble,65000,7516171.00

bubble,75000,10042344.00

bubble,85000,12926647.00

bubble,125000,28169823.00

bubble,55000,5330411.00

bubble,80000,11461316.00

bubble,90000,14503559.00

bubble,5000,24566.77

bubble,10000,132262.38

bubble,15000,329427.50

bubble,40000,2738169.00

bubble,50000,4375322.00

bubble,95000,16248882.00

heap,25000,339932.67

heap,150000,12253064.00

heap,160000,13923912.00

heap,20000,218975.00

heap,195000,20667508.00

heap,70000,2660426.00

heap,105000,6006324.00

heap,140000,10675954.00

heap,45000,1100479.00

heap,65000,2301011.00

heap,75000,3057205.00

heap,85000,3926421.00

heap,125000,8487344.00

heap,175000,16649700.00

heap,185000,18620458.00

heap,200000,21779415.00

heap,5000,13780.42

heap,10000,54680.29

heap,15000,122578.78

heap,40000,870371.50

heap,50000,1358327.00

heap,55000,1645083.00

heap,80000,3482939.00

heap,90000,4398030.00

heap,95000,4905566.00

heap,35000,670095.00

heap,115000,7217173.00

heap,130000,9196417.00

heap,135000,9897388.00

heap,165000,14825167.00

heap,30000,491160.67

heap,60000,1954291.00

heap,120000,7849452.00

heap,145000,11433287.00

heap,155000,13118703.00

heap,100000,5435468.00

heap,110000,6581084.00

heap,170000,15735724.00

heap,180000,17653037.00

heap,190000,19649447.00

insertion,135000,3065212.00

insertion,165000,4603031.00

insertion,35000,200138.50

insertion,115000,2226400.00

insertion,130000,2846580.00

insertion,145000,3546486.00

insertion,155000,4068278.00

insertion,30000,147193.14

insertion,60000,594780.00

insertion,120000,2423948.00

insertion,180000,5445186.00

insertion,190000,6084636.00

insertion,100000,1670105.00

insertion,110000,2024639.00

insertion,170000,4852704.00

insertion,25000,101852.40

insertion,150000,3820168.00

insertion,160000,4297208.00

insertion,20000,65396.17

insertion,195000,6431371.00

insertion,70000,804843.00

insertion,105000,1853340.00

insertion,140000,3287070.00

insertion,85000,1206702.00

insertion,125000,2612560.00

insertion,175000,5158088.00

insertion,185000,5791243.00

insertion,200000,6730165.00

insertion,45000,329992.25

insertion,65000,702183.50

insertion,75000,938237.00

insertion,40000,261509.50

insertion,50000,414540.33

insertion,55000,502956.00

insertion,80000,1057163.00

insertion,90000,1356339.00

insertion,5000,4162.43

insertion,10000,16384.14

insertion,15000,37104.62

insertion,95000,1510717.00

radix,20000,4027.61

radix,195000,39729.07

radix,140000,28308.29

radix,70000,14112.45

radix,105000,21139.62

radix,75000,15111.05

radix,85000,17209.72

radix,125000,25185.22

radix,175000,35497.06

radix,185000,37550.68

radix,200000,40738.43

radix,45000,9086.67

radix,65000,13080.65

radix,15000,3009.89

radix,40000,8052.30

radix,50000,10094.92

radix,55000,11060.91

radix,80000,16215.18

radix,90000,18178.21

radix,5000,1002.44

radix,10000,2011.51

radix,95000,19254.53

radix,130000,26222.55

radix,135000,27247.67

radix,165000,33434.97

radix,35000,7027.71

radix,115000,23352.88

radix,120000,24170.09

radix,145000,29393.16

radix,155000,31482.24

radix,30000,6022.31

radix,60000,12045.05

radix,170000,34423.91

radix,180000,36935.72

radix,190000,38506.13

radix,100000,20137.46

radix,110000,22159.63

radix,160000,32300.09

radix,25000,5044.18

radix,150000,30352.36

selection,100000,5883260.00

selection,110000,7127927.00

selection,170000,16995820.00

selection,180000,19074723.00

selection,190000,21240679.00

selection,25000,368271.00

selection,150000,13224163.00

selection,160000,15073359.00

selection,20000,235758.40

selection,195000,22359061.00

selection,70000,2895208.00

selection,105000,6501675.00

selection,140000,11538947.00

selection,175000,18020188.00

selection,185000,20138759.00

selection,200000,23527233.00

selection,45000,1199787.00

selection,65000,2488895.00

selection,75000,3322807.00

selection,85000,4249546.00

selection,125000,9207688.00

selection,55000,1783854.00

selection,80000,3780673.00

selection,90000,4759257.00

selection,5000,14888.40

selection,10000,59080.47

selection,15000,132470.00

selection,40000,943192.00

selection,50000,1474782.00

selection,95000,5306705.00

selection,35000,721570.50

selection,115000,7797785.00

selection,130000,9923183.00

selection,135000,10730160.00

selection,165000,16025007.00

selection,30000,529710.50

selection,60000,2121569.00

selection,120000,8471901.00

selection,145000,12420929.00

selection,155000,14136060.00

shell,75000,131071.25

shell,85000,154296.29

shell,125000,336583.67

shell,175000,687707.50

shell,185000,756006.50

shell,200000,1058086.00

shell,45000,46274.64

shell,65000,95353.58

shell,15000,5636.38

shell,40000,44423.60

shell,50000,62233.33

shell,55000,68769.47

shell,80000,173632.67

shell,90000,193987.50

shell,5000,751.42

shell,10000,2922.33

shell,95000,199056.67

shell,130000,409055.67

shell,135000,407958.00

shell,165000,586413.00

shell,35000,29189.53

shell,115000,302485.75

shell,120000,383151.67

shell,145000,463199.33

shell,155000,553247.50

shell,30000,22832.73

shell,60000,93361.58

shell,170000,699188.00

shell,180000,836425.50

shell,190000,876609.50

shell,100000,256315.00

shell,110000,295942.00

shell,160000,701247.00

shell,25000,14518.09

shell,150000,556246.50

shell,20000,11044.47

shell,195000,871387.50

shell,140000,504436.50

shell,70000,121063.44

shell,105000,245085.60

quick,115000,9668.37

quick,130000,11212.42

quick,135000,11731.14

quick,165000,14545.52

quick,35000,2689.38

quick,60000,4760.96

quick,120000,10205.28

quick,145000,12443.99

quick,155000,13481.72

quick,30000,2312.73

quick,110000,9400.26

quick,170000,14829.55

quick,180000,15732.44

quick,190000,16921.16

quick,100000,8393.45

quick,150000,12953.02

quick,160000,13803.03

quick,25000,1854.87

quick,195000,17219.56

quick,20000,1454.58

quick,105000,8816.42

quick,140000,12163.54

quick,70000,5772.07

quick,65000,5281.44

quick,75000,6144.47

quick,85000,7032.49

quick,125000,10762.37

quick,175000,15745.57

quick,185000,16540.93

quick,200000,17820.80

quick,45000,3614.82

quick,10000,680.14

quick,15000,1059.26

quick,40000,3100.83

quick,50000,3964.48

quick,55000,4362.43

quick,80000,6551.29

quick,90000,7442.16

quick,5000,321.38

quick,95000,7927.38

merge,30000,4152.66

merge,60000,8743.74

merge,120000,18318.94

merge,145000,22404.69

merge,155000,24144.15

merge,100000,15151.38

merge,110000,16717.69

merge,170000,26873.76

merge,180000,28649.40

merge,190000,30169.21

merge,25000,3454.08

merge,150000,23554.02

merge,160000,25177.76

merge,20000,2709.00

merge,195000,30929.98

merge,70000,10503.00

merge,105000,16191.33

merge,140000,22044.30

merge,175000,28269.95

merge,185000,29920.40

merge,200000,32380.17

merge,45000,6547.97

merge,65000,9623.68

merge,75000,11459.40

merge,85000,13091.42

merge,125000,19586.70

merge,55000,8146.79

merge,80000,12164.75

merge,90000,13875.24

merge,5000,614.32

merge,10000,1301.52

merge,15000,2021.10

merge,40000,5843.77

merge,50000,7457.76

merge,95000,14731.92

merge,35000,5067.56

merge,115000,18152.68

merge,130000,20400.08

merge,135000,21598.73

merge,165000,26791.64

bubble,85000,12876696.00

bubble,125000,28164932.00

bubble,175000,54964847.00

bubble,185000,61572446.00

bubble,200000,71921101.00

bubble,45000,3493667.00

bubble,65000,7529996.00

bubble,75000,10054434.00

bubble,40000,2735764.00

bubble,50000,4373558.00

bubble,55000,5357295.00

bubble,80000,11443947.00

bubble,90000,14531618.00

bubble,5000,24682.44

bubble,10000,131887.88

bubble,15000,337910.33

bubble,95000,16790187.00

bubble,135000,34922290.00

bubble,165000,49609444.00

bubble,35000,2096648.00

bubble,115000,23982084.00

bubble,130000,30544139.00

bubble,145000,39039686.00

bubble,155000,43434336.00

bubble,30000,1514157.00

bubble,60000,6430559.00

bubble,120000,26095840.00

bubble,180000,58360860.00

bubble,190000,65274548.00

bubble,100000,18035632.00

bubble,110000,21770348.00

bubble,170000,52096922.00

bubble,25000,1015244.00

bubble,150000,40655113.00

bubble,160000,46199998.00

bubble,20000,626035.00

bubble,195000,68605629.00

bubble,70000,8727554.00

bubble,105000,19895464.00

bubble,140000,35217795.00

heap,55000,1647603.00

heap,80000,3483381.00

heap,90000,4419759.00

heap,5000,13811.27

heap,10000,54819.19

heap,15000,123235.89

heap,40000,871669.00

heap,50000,1357161.00

heap,95000,4901497.00

heap,35000,667497.00

heap,115000,7205645.00

heap,130000,9186763.00

heap,135000,9911841.00

heap,165000,14847391.00

heap,30000,488818.33

heap,60000,1960650.00

heap,120000,7838688.00

heap,145000,11425205.00

heap,155000,13059351.00

heap,100000,5432842.00

heap,110000,6595514.00

heap,170000,15732999.00

heap,180000,17626412.00

heap,190000,19641237.00

heap,25000,360655.67

heap,150000,12225561.00

heap,160000,13920040.00

heap,20000,217912.00

heap,195000,20686446.00

heap,70000,2673907.00

heap,105000,6007742.00

heap,140000,10651590.00

heap,175000,16657611.00

heap,185000,18664557.00

heap,200000,21746073.00

heap,45000,1103451.00

heap,65000,2296115.00

heap,75000,3064990.00

heap,85000,3925556.00

heap,125000,8492378.00

insertion,25000,101814.90

insertion,150000,3815342.00

insertion,160000,4299912.00

insertion,20000,65738.17

insertion,195000,6434032.00

insertion,70000,804402.50

insertion,105000,1854777.00

insertion,140000,3281470.00

insertion,85000,1210498.00

insertion,125000,2621827.00

insertion,175000,5167325.00

insertion,185000,5813213.00

insertion,200000,6721123.00

insertion,45000,335066.00

insertion,65000,700456.50

insertion,75000,933786.50

insertion,40000,265855.75

insertion,50000,413262.00

insertion,55000,501740.50

insertion,80000,1074067.00

insertion,90000,1361675.00

insertion,5000,4159.91

insertion,10000,16342.95

insertion,15000,37282.28

insertion,95000,1518200.00

insertion,135000,3066306.00

insertion,165000,4563943.00

insertion,35000,201714.80

insertion,115000,2231897.00

insertion,130000,2834259.00

insertion,145000,3539525.00

insertion,155000,4037002.00

insertion,30000,147392.71

insertion,60000,593760.50

insertion,120000,2420003.00

insertion,180000,5451917.00

insertion,190000,6079009.00

insertion,100000,1685217.00

insertion,110000,2031647.00

insertion,170000,4858591.00

radix,25000,5041.07

radix,150000,30330.61

radix,160000,32412.94

radix,20000,4022.98

radix,195000,39628.47

radix,70000,14098.80

radix,105000,21212.52

radix,140000,28433.45

radix,45000,9061.80

radix,65000,13079.55

radix,75000,15184.09

radix,85000,17359.30

radix,125000,25334.63

radix,175000,35614.00

radix,185000,37681.55

radix,200000,40968.46

radix,5000,1000.42

radix,10000,2009.70

radix,15000,3014.73

radix,40000,8069.93

radix,50000,10108.30

radix,55000,11149.63

radix,80000,16152.47

radix,90000,18196.12

radix,95000,19243.08

radix,35000,7053.60

radix,115000,23245.04

radix,130000,26223.73

radix,135000,27288.79

radix,165000,33408.62

radix,30000,6045.53

radix,60000,12103.45

radix,120000,24271.38

radix,145000,29585.17

radix,155000,31463.64

radix,100000,20232.78

radix,110000,22176.85

radix,170000,34410.91

radix,180000,36692.31

radix,190000,38671.29

selection,5000,14901.80

selection,10000,59588.05

selection,15000,132863.62

selection,40000,941905.50

selection,50000,1491283.00

selection,55000,1797820.00

selection,80000,3766067.00

selection,90000,4755962.00

selection,95000,5306617.00

selection,35000,721186.50

selection,115000,7794027.00

selection,130000,9929957.00

selection,135000,10727525.00

selection,165000,16022712.00

selection,30000,530955.50

selection,60000,2126826.00

selection,120000,8460424.00

selection,145000,12367582.00

selection,155000,14134418.00

selection,100000,5879686.00

selection,110000,7113314.00

selection,170000,17041625.00

selection,180000,19055086.00

selection,190000,21254362.00

selection,25000,366624.33

selection,150000,13212105.00

selection,160000,15067656.00

selection,20000,235430.20

selection,195000,22365249.00

selection,70000,2892561.00

selection,105000,6518146.00

selection,140000,11515350.00

selection,45000,1203212.00

selection,65000,2488066.00

selection,75000,3319084.00

selection,85000,4253564.00

selection,125000,9189913.00

selection,175000,18002392.00

selection,185000,20113462.00

selection,200000,23511827.00

insertion,5000,4173.40

insertion,10000,16338.89

insertion,15000,36975.36

insertion,40000,264159.50

insertion,50000,413010.33

insertion,55000,501857.50

insertion,80000,1066085.00

insertion,90000,1369896.00

insertion,95000,1502831.00

insertion,35000,201917.00

insertion,115000,2229072.00

insertion,130000,2830820.00

insertion,135000,3085603.00

insertion,165000,4571308.00

insertion,30000,147128.71

insertion,60000,594421.50

insertion,120000,2410502.00

insertion,145000,3516864.00

insertion,155000,4085304.00

insertion,100000,1659131.00

insertion,110000,2030263.00

insertion,170000,4860253.00

insertion,180000,5403079.00

insertion,190000,6126207.00

insertion,25000,102062.30

insertion,150000,3814816.00

insertion,160000,4311492.00

insertion,20000,65427.12

insertion,195000,6444928.00

insertion,70000,807095.50

insertion,105000,1854327.00

insertion,140000,3270046.00

insertion,45000,333675.00

insertion,65000,699923.00

insertion,75000,940172.50

insertion,85000,1205295.00

insertion,125000,2610082.00

insertion,175000,5152316.00

insertion,185000,5796484.00

insertion,200000,6719244.00

radix,130000,26335.28

radix,135000,27386.26

radix,165000,33530.97

radix,35000,7060.81

radix,115000,23281.20

radix,120000,24192.33

radix,145000,29304.22

radix,155000,31478.53

radix,30000,6029.29

radix,60000,12147.37

radix,170000,34631.61

radix,180000,36451.22

radix,190000,38559.54

radix,100000,20275.04

radix,110000,22308.83

radix,160000,32274.73

radix,25000,5024.63

radix,150000,30401.97

radix,20000,4031.21

radix,195000,39662.54

radix,140000,28470.22

radix,70000,14101.33

radix,105000,21161.20

radix,75000,15105.14

radix,85000,17309.24

radix,125000,25226.49

radix,175000,35710.27

radix,185000,37466.29

radix,200000,40854.04

radix,45000,9100.41

radix,65000,13086.34

radix,15000,3009.79

radix,40000,8056.53

radix,50000,10100.07

radix,55000,11079.17

radix,80000,16173.49

radix,90000,18134.95

radix,5000,999.72

radix,10000,2012.94

radix,95000,19126.84

selection,40000,938137.00

selection,50000,1476980.00

selection,55000,1778211.00

selection,80000,3768856.00

selection,90000,4763369.00

selection,5000,14902.48

selection,10000,59136.61

selection,15000,132538.25

selection,95000,5325552.00

selection,135000,10726972.00

selection,165000,16015340.00

selection,35000,722215.00

selection,115000,7789948.00

selection,130000,9941897.00

selection,145000,12352601.00

selection,155000,14139590.00

selection,30000,528578.00

selection,60000,2124211.00

selection,120000,8443330.00

selection,180000,19047698.00

selection,190000,21244423.00

selection,100000,5891011.00

selection,110000,7131636.00

selection,170000,16996914.00

selection,25000,367408.33

selection,150000,13225382.00

selection,160000,15071509.00

selection,20000,234902.00

selection,195000,22363673.00

selection,70000,2889168.00

selection,105000,6490755.00

selection,140000,11572639.00

selection,85000,4249473.00

selection,125000,9203176.00

selection,175000,18018507.00

selection,185000,20138271.00

selection,200000,23543098.00

selection,45000,1190101.00

selection,65000,2489209.00

selection,75000,3307581.00

shell,110000,296035.25

shell,170000,698824.50

shell,180000,835541.00

shell,190000,876876.00

shell,100000,256709.25

shell,150000,557004.50

shell,160000,700248.00

shell,25000,14508.26

shell,195000,871417.00

shell,20000,11037.72

shell,105000,244369.40

shell,140000,502797.00

shell,70000,120981.33

shell,65000,95143.25

shell,75000,131028.50

shell,85000,154738.71

shell,125000,336035.00

shell,175000,684589.00

shell,185000,754341.00

shell,200000,1064626.00

shell,45000,46337.12

shell,10000,2918.68

shell,15000,5645.75

shell,40000,44263.88

shell,50000,62528.63

shell,55000,68747.94

shell,80000,173606.50

shell,90000,193923.50

shell,5000,751.05

shell,95000,199473.67

shell,115000,300898.00

shell,130000,411493.00

shell,135000,408273.00

shell,165000,581690.50

shell,35000,29221.33

shell,60000,94100.42

shell,120000,383280.00

shell,145000,466603.33

shell,155000,555010.50

shell,30000,22773.48

quick,135000,11623.02

quick,165000,14382.73

quick,35000,2665.31

quick,115000,9645.08

quick,130000,11153.49

quick,145000,12444.12

quick,155000,13525.35

quick,30000,2309.61

quick,60000,4742.37

quick,120000,10165.91

quick,180000,16049.29

quick,190000,16703.62

quick,100000,8535.99

quick,110000,9360.68

quick,170000,14876.67

quick,25000,1857.70

quick,150000,12913.76

quick,160000,14207.79

quick,20000,1456.77

quick,195000,17116.30

quick,70000,5755.08

quick,105000,8799.29

quick,140000,12119.33

quick,85000,6952.51

quick,125000,10722.41

quick,175000,15484.53

quick,185000,16374.97

quick,200000,18125.76

quick,45000,3562.83

quick,65000,5348.94

quick,75000,6139.68

quick,40000,3122.33

quick,50000,3973.32

quick,55000,4352.03

quick,80000,6533.34

quick,90000,7344.25

quick,5000,317.33

quick,10000,675.93

quick,15000,1060.84

quick,95000,7890.39

merge,70000,10793.11

merge,105000,16642.56

merge,140000,22469.56

merge,185000,30471.65

merge,200000,33111.97

merge,45000,6719.20

merge,65000,9918.65

merge,75000,11659.91

merge,85000,13319.39

merge,125000,19974.97

merge,175000,29217.84

merge,80000,12549.43

merge,90000,14195.87

merge,5000,628.00

merge,10000,1330.73

merge,15000,2067.36

merge,40000,5988.53

merge,50000,7660.66

merge,55000,8422.36

merge,95000,15078.59

merge,35000,5189.62

merge,115000,18561.81

merge,130000,20882.62

merge,135000,22228.41

merge,165000,27565.51

merge,30000,4408.49

merge,60000,9305.94

merge,120000,19523.49

merge,145000,23983.46

merge,155000,26086.13

merge,100000,16206.67

merge,110000,17961.30

merge,170000,28762.33

merge,180000,30255.55

merge,190000,31999.24

merge,25000,3706.37

merge,150000,25099.69

merge,160000,26781.51

merge,20000,2882.00

merge,195000,32817.00

bubble,65000,7489509.00

bubble,75000,10047057.00

bubble,85000,12919699.00

bubble,125000,28052547.00

bubble,175000,54919944.00

bubble,185000,61679825.00

bubble,200000,71764890.00

bubble,45000,3502515.00

bubble,10000,131713.50

bubble,15000,329797.25

bubble,40000,2740376.00

bubble,50000,4366577.00

bubble,55000,5349854.00

bubble,80000,11461327.00

bubble,90000,14506032.00

bubble,5000,24601.67

bubble,95000,16235787.00

bubble,115000,23816161.00

bubble,130000,30348198.00

bubble,135000,32874680.00

bubble,165000,48741485.00

bubble,35000,2055313.00

bubble,60000,6382592.00

bubble,120000,25962973.00

bubble,145000,37766124.00

bubble,155000,43288312.00

bubble,30000,1479963.00

bubble,110000,21731134.00

bubble,170000,51916495.00

bubble,180000,58094809.00

bubble,190000,64987828.00

bubble,100000,17918851.00

bubble,150000,40505731.00

bubble,160000,46004671.00

bubble,25000,1014206.00

bubble,195000,68538503.00

bubble,20000,622806.00

bubble,105000,19890885.00

bubble,140000,35213391.00

bubble,70000,8753071.00

heap,20000,218633.60

heap,195000,20690598.00

heap,70000,2659251.00

heap,105000,5984577.00

heap,140000,10638149.00

heap,200000,21779050.00

heap,45000,1098974.00

heap,65000,2300117.00

heap,75000,3076735.00

heap,85000,3927147.00

heap,125000,8492019.00

heap,175000,16634378.00

heap,185000,18636008.00

heap,90000,4401870.00

heap,5000,13821.16

heap,10000,54727.95

heap,15000,122721.89

heap,40000,869312.50

heap,50000,1355457.00

heap,55000,1641784.00

heap,80000,3499628.00

heap,95000,4911425.00

heap,35000,667032.00

heap,115000,7195636.00

heap,130000,9210732.00

heap,135000,9909776.00

heap,165000,14805338.00

heap,30000,489332.67

heap,60000,1968664.00

heap,120000,7815641.00

heap,145000,11433316.00

heap,155000,13056051.00

heap,100000,5431301.00

heap,110000,6579085.00

heap,170000,15710343.00

heap,180000,17681716.00

heap,190000,19607151.00

heap,25000,341432.33

heap,150000,12229860.00

heap,160000,13904363.00