**CAPA**

# 1. Introdução

O problema proposto foi a implementação de um "Ordenador Universal", que seria um ordenador "inteligente" e capaz de determinar, por si próprio, parâmetros para a minimização do custo da ordenação de estruturas de entrada, sendo utilizados, na ordenação, os algoritmos de Insertion Sort e Quick Sort (otimizado com mediana de 3 e Insertion para partições pequenas). Para tanto, o ordenador determina, baseando-se em fatores de custo pré-determinados e na configuração da entrada, os valores do Limiar de Quebras (que indica para qual número de quebras na entrada o Insertion é automaticamente menos custoso) e Limiar de Partição (que indica a partir de qual tamanho de partição devemos chamar o Insertion dentro do próprio Quick Sort), permitindo ordená-la com o menor custo possível seguindo o limite de custo desejado.

Esta documentação tem como objetivo descrever a implementação do TAD criado para a solução do problema (o que é feito na seção 2, "Implementação", e na 4, "Estratégias de Robustez") e realizar a análise de seu funcionamento e possíveis custos gerados (o que é feito na seção 3, "Análise de Complexidade", na 5, "Análise Experimental" e na 6, "Conclusões"). A seção 7 conta com a bibliografia utilizada no trabalho.

# 2. Método/Implementação

O programa foi desenvolvido em linguagem C++ e compilado com o compilador g++ da GNU Compiler Collection, especificamente na versão c++11.

O código está organizado em, além do Makefile, mais 5 arquivos, sendo 2 deles de tipo .hpp e 3 de tipo .cpp. O arquivo Ordenadores.cpp faz a implementação dos algoritmos Quick Sort e Insertion Sort como eles são utilizados pelo TAD Ordenador Universal, além da struct contador_t e dos métodos que lidam com ela. Em OrdenadorUniversal.cpp, temos a implementação da classe ordUniversal, que é o TAD em si. Em main.cpp, temos a interação com a entrada padrão e a chamada dos métodos necessários do TAD.

Para solucionar o problema, foram implementadas 3 structs e uma classe. A struct contador_t é utilizada para registrar os valores de comparações, chamadas e movimentações nos algoritmos de ordenação e no método ordenadorUniversal. A struct estatisticas_t é o tipo utilizado para o registro de estatísticas no método determinaLimiarParticao e tem como membros, além do valor de custo e do limiar de partição que devem ser armazenados nesse método, uma instância de contador_t. A struct estatisticasLQ tem estrutura similar e é utilizada para registrar as estatísticas necessárias para obter o limiar de quebras. Ela conta com um membro diffCusto, que armazena a diferença de custos entre o Insertion e o QuickSort, um campo para o valor do limiar de quebras, 2 instâncias de contador_t e 2 valores de custo, que armazenam separadamente os dados para a execução do Insertion e do Quicksort. A decisão de implementar a struct desta forma foi tomada com o fim de reduzir a

quantidade de adaptações necessárias no método CalculaNovaFaixa para obter a nova faixa de limiares de quebras. Por fim, a classe ordUniversal implementa de fato o TAD Ordenador Universal. Os principais métodos da classe são ordenadorUniversal, determinaLimiarParticao, CalculaNovaFaixa, determinaLimiarQuebras e CalculaNovaFaixaLQ. Os demais são métodos auxiliares que são utilizados por eles ou para obter e registrar informações sobre o TAD.

O método ordenadorUniversal traz a funcionalidade do TAD em si, definindo qual deve ser o algoritmo utilizado com base na quantidade de quebras da entrada e nos valores de limiar de quebras e limiar de partição, obtidos chamando os métodos correspondentes anteriormente.

O método determinaLimiarParticao tem como objetivo encontrar o limiar de partição a partir do qual devemos chamar o Insertion dentro do próprio Quick Sort. Para tanto, o método testa diferentes valores possíveis de limiares (limParticao), obtém os custos associados a cada valor de limiar, a diferença de custo entre minMPS e maxMPS (os extremos da faixa de valores possíveis) e refina a faixa para cálculo de limiares. Esse processo continua até que a diferença de custo obtida seja menor que o limiar de custo pré-determinado ou até que o número de possíveis partições obtido em uma dada iteração seja menor que 5. A primeira faixa testada vai de 2 a tam, que é o tamanho do vetor. Em cada iteração seguinte, dividimos a faixa em 5 intervalos equidistantes, é feita uma cópia do vetor de entrada, que é ordenada utilizando o método ordenadorUniversal, e são registradas, no array de estatísticas (estatisticas), o custo e os valores de comparações, movimentações e chamadas para cada um dos valores de partição testados. Após serem completadas todas as iterações (sendo que são sempre, no máximo, 6), é encontrado o índice no vetor que armazena o menor deles com o método menorCusto. É chamado o método CalculaNovaFaixa, que obtém os novos limites minMPS e maxMPS para a próxima iteração e, por fim, é obtida a diferença de custo (diffCusto) entre os limites da nova faixa. Ao fim do processo, quando alguma das condições iniciais falha, é retornado o valor final obtido para o limiar de partição. O método registraEstatisticas calcula o custo com base nos dados armazenados em contador_t, e o imprimeEstatisticas faz a impressão dos dados obtidos a cada iteração. Por fim, o método getMPS, utilizado em CalculaNovaFaixa, obtém o valor de limiar de partição armazenado em um índice específico do vetor.

O método determinaLimiarQuebras segue uma lógica parecida com o determinaLimiarParticao, tendo o objetivo de encontrar o limiar de quebras para o qual é menos custoso já chamar o Insertion Sort ao invés do Quick Sort. O método cria uma cópia do vetor de entrada, o ordena utilizando Quick Sort e, com o método shuffleVector, cria no vetor já ordenado a quantidade de quebras desejadas para cada teste. É feita uma nova cópia desse vetor e, então, ordenamos uma das cópias com o Insertion Sort e uma delas com o Quick Sort. Os custos (e comparações, chamadas e movimentações) da ordenação usando cada um dos algoritmos são registrados no vetor de estatísticas statsLQ. Quando finalizamos todas as iterações, o método calculaDiffCustosLQ calcula a diferença de custo entre Insertion e Quick Sort para

cada um dos valores de limiar de quebras testados. É obtido o índice no vetor da menor dentre essas diferenças de custo com o método menorCustoLQ, que é utilizado em calculaNovaFaixaLQ para obter a nova faixa de valores de quebras a serem testados. Assim, registramos o valor de diffCusto, correspondente à diferença de custo entre o maior e o menor custo de ordenação utilizando o Insertion Sort. O processo continua até que diffCusto seja menor que o limiar de custo pré-determinado ou que a quantidade de valores para limiares de quebras obtido seja menor que 5. Assim, é retornado o valor final obtido para o limiar de quebras. O método registraEstatisticas é o mesmo do utilizado em determinaLimiarParticao e calcula os custos com base nos coeficientes a, b, c e nas estatísticas de movimentações, comparações e chamadas, e o imprimeEstatisticasLQ imprime os dados obtidos. O método getLQ, utilizado em calculaNovaFaixaLQ, obtém o valor do limiar de quebras armazenado em um dado índice do vetor de estatísticas. Um ponto importante é que, como o valor de numLQ (quantidade de valores de limiar de quebras em uma faixa) não é fixo (como em numMPS), calculamos previamente qual será o valor de numLQ para alocar o vetor de estatisticas (estatisticasLQ) corretamente. Além disso, também temos, na classe ordUniversal, o construtor da classe e o método calculaQuebras, que conta o número de quebras no vetor de entrada.

O fluxo do programa é definido por main.cpp. É feita a leitura dos dados do arquivo de entrada, com tratamento para casos de erro relacionados à leitura, inicializada uma instância da classe com os dados lidos (coeficientes a, b, c, limiar de custo e seed) e, então, chamados os métodos determinaLimiarParticao e determinaLimiarQuebras. Esses valores também são registrados na instância usando os métodos set.

## 3. Análise de Complexidade

Consideramos n como o tamanho do vetor de entrada e, portanto, o tamanho do problema. Faremos a análise dos métodos auxiliares e, posteriormente, dos métodos que os utilizam.

1. **calculaQuebras:** o método itera de 0 a n-1 e realiza uma comparação e, possivelmente, uma operação de custo constante a cada iteração. Logo, a sua complexidade de tempo é $O(n)$. O vetor analisado, de tamanho n, é recebido como parâmetro pelo método e são utilizadas apenas estruturas auxiliares unitárias $O(1)$. Com isso, a complexidade de espaço é Θ(1).
2. **getMPS e getLQ:** os métodos fazem uma única comparação (para checar possíveis erros de acesso a posições indevidas do vetor) e uma operação de custo constante (return). Logo, sua complexidade de tempo é $O(1)$. Não há uso de memória extra, com complexidade de espaço $O(1)$.
3. **menorCusto e menorCustoLQ:** os métodos iteram sobre os vetores de estatísticas usados em determinaLimiarParticao e determinaLimiarQuebras, respectivamente. Sabe-se que o tamanho desses vetores não depende

diretamente de n e, no for, a cada iteração é realizada uma comparação e possivelmente apenas operações de custo constante, como atribuições. Assim, a complexidade de tempo é $O(1)$. Também não há uso de memória extra, com complexidade de espaço $O(1)$.

4. **encontraElemento e encontraElementoLQ:** os métodos iteram sobre os vetores de estatísticas usados em determinaLimiarParticao e determinaLimiarQuebras, indo de 0 a numMPS-1 e de 0 a numLQ-1, respectivamente. É realizada uma comparação e possivelmente uma operação de custo constante a cada iteração do for. Como sabemos que numMPS e numLQ não são funções de n e, em geral, não assumem valores muito grandes, a complexidade de tempo é $O(1)$. Já a complexidade de espaço é $O(1)$, já que não há uso de estruturas de memória auxiliares.
5. **registraEstatisticas:** o método realiza somente operações de custo constante (atribuição, operações matemáticas) e uma única vez, com complexidade de tempo $O(1)$. Já a complexidade de espaço é $O(1)$, uma vez que não é utilizada memória auxiliar.
6. **imprimeEstatisticas e imprimeEstatisticasLQ:** os métodos apenas fazem impressões de dados, com complexidade de tempo $O(1)$. Não há uso de memória auxiliar, com complexidade de tempo $O(1)$.
7. **calculaDiffCustosLQ:** iteramos de 0 a numLQ-1, realizando apenas operações de custo constante a cada iteração. Com isso, a complexidade de tempo é $O(1)$. O vetor analisado é recebido por parâmetro e não há alocação de novas variáveis ou vetores, com complexidade de espaço $O(1)$.
8. **shuffleVector:** o método itera de 0 a numShuffle-1, realizando somente operações de custo constante a cada iteração. Com isso, sua complexidade de tempo é $O(numShuffle)$. Como o vetor utilizado é passado por parâmetro e há somente alocação de 3 variáveis, a complexidade de espaço é $O(1)$.
9. **calculaNovaFaixa:** o método CalculaNovaFaixa realiza uma quantidade constante de operações de custo também constante (comparações, atribuições e operações aritméticas). Além disso, a única função que chama é getMPS, cujos custos são também constantes. Com isso, a complexidade de tempo é $O(1)$. Como há alocação unicamente de 2 variáveis de tipo int, a complexidade de espaço é $O(1)$.
10. **calculaNovaFaixaLQ:** similarmente a calculaNovaFaixa, realiza uma quantidade constante de comparações e operações de custo constante, além de não utilizar estruturas extras de memória. Com isso, a complexidade de tempo é $O(1)$ e a complexidade de espaço é, também, $O(1)$.
11. **ordenadorUniversal:** para este método, analisamos os casos possíveis de execução. Se nroQuebras < limQuebras, sabemos que o Insertion é eficiente e apresenta complexidade $O(n)$. Se esse não é o caso, seguimos para a chamada do quicksort, já otimizado, que apresenta complexidade de tempo $O(nlogn)$.

Já para a complexidade de espaço, há o uso de memória extra na pilha de chamadas recursivas para o QuickSort, com complexidade $O(nlogn)$.

12. **determinaLimiarParticao:** é o método mais "problemático", uma vez que utiliza grande parte dos demais métodos e, além disso, conta com um for aninhado com um while. A quantidade de vezes que o for roda não depende necessariamente de n e nunca é maior do que 6, ou seja, executa uma quantidade aproximadamente constante de vezes. Em cada iteração do for, são chamadas os métodos de custo constante resetcounter, registraEstatisticas e imprimeEstatisticas. Além disso, é chamado o ordenadorUniversal que, da forma como é utilizado no método, chama sempre somente o QuickSort, com custo $O(nlogn)$. Ainda dentro do while, são chamados os métodos calculaNovaFaixa, menorCusto e encontraElemento, todos de custo constante, de forma que o desafio é determinar quantas vezes o while roda. Notamos que, a cada nova faixa, utilizamos ¾ do intervalo obtido anteriormente. Com isso, a cada iteração do while, eliminamos ¼ da faixa analisada, executando cerca de $log(n)$ vezes. Com isso, a complexidade de tempo é de $O(nlogn * log(n))$, ou seja, $O(n * log^2(n))$ Já para a complexidade de espaço, a cada iteração fazemos uma única cópia do vetor de entrada para performar a ordenação e, ao final de cada iteração, esse vetor é desalocado. Assim, a quantidade de memória extra utilizada (contando com o vetor de estatísticas, de tamanho constante) nunca extrapola o valor de n, com complexidade de espaço $O(n)$.
13. **determinaLimiarQuebras:** a análise de complexidade para este método é próxima da análise realizada para determinaLimiarParticao. O for executa um número de vezes que é independente de n e consideramos constante. Os métodos chamados em todo o método, com exceção do QuickSort e Insertion Sort, apresentam custo constante. Em cada iteração do for, chamamos o quicksort, com custo nlogn, e o Insertion, com custo $O(n^2)$, gerando um custo de $O(n^2)$ para o for. Para o while, também reduzimos cada vez em ¾ do tamanho, executando cerca de log(n) vezes. Com isso, é produzida uma complexidade de tempo de $O(n^2 * log(n))$. Já para o espaço, a cada iteração do for, alocamos 2 vetores de tamanho n para as cópias do vetor de entrada, que são desalocados ao fim do for. Com isso, nunca é utilizada memória extra que extrapole o valor de 2*n, gerando uma complexidade de espaço de $O(n)$.

## 4. Estratégias de Robustez

A principal estratégia de robustez empregada foi o tratamento de exceções utilizando o método de try-catch-throw disponível na linguagem C++. Em especial, foi utilizado em main.cpp (o arquivo que de fato interage com a entrada padrão) com o fim de indicar erros na leitura do arquivo de entrada (evidenciando se houve algum problema na leitura ou se a entrada estava formatada indevidamente, por exemplo) ou

no recebimento do arquivo pela entrada padrão. Assim, caso ocorram erros nessa parte do sistema, eles são bem identificados, não se propagam e não geram comportamentos inesperados do programa. Além disso, também foi utilizado o tratamento de exceções nos momentos em que é utilizada a alocação dinâmica de memória (para verificar se foi bem-sucedida) e, em geral, nos métodos que poderiam possivelmente acessar posições inválidas em vetores (notadamente por terem os índices de acesso calculados por outros métodos no programa).

Por fim, também foi feita a modularização das classes empregadas. Os seus atributos principais são mantidos privados e, portanto, inacessíveis a alterações diretas e possivelmente maliciosas por parte de usuários.

# 5. Análise Experimental

## 5.1. Método e Resultados

Na primeira parte da análise, para obter os coeficiente a, b e c, fixamos os valores do tamanho de chave e do registro em 4 bytes (correspondentes ao tamanho de um int) e a configuração do vetor em próximo a medianamente ordenado, contando sempre com metade de seu tamanho em número de quebras. Os valores de tamanho de vetor empregados iniciaram em 512 e dobraram a cada teste, até que se chegasse ao tamanho de 32768. Uma vez que não é possível utilizar o ordenadorUniversal sem antes definir os limiares, o algoritmo utilizado para cálculo dos coeficientes foi o QuickSort com mediana de 3 e uso de Insertion Sort para partições menores que 50. Por fim, como a cada execução obtemos um tempo de execução levemente diferente, foram realizadas 5 execuções para cada caso e o tempo final empregado foi a média do tempo para essas execuções. Após a coleta desses dados, foi realizada a Regressão Linear com os dados da Tabela 1.

Com a calibragem de a, b e c, foram variados os demais parâmetros que estavam fixos antes (tamanho da chave, do registro e ordenação do vetor), sendo obtidos os resultados mostrados na Tabela 2. Para auxiliar na interpretação e visualização dos dados apresentados, também foram criados os gráficos 1 e 2.

| Tam. Vetor | cmp | mv | calls | tempo | a | b | c |
|---|---|---|---|---|---|---|---|
| 512 | 2536 | 1774 | 46 | 0,0000284 | - | - | - |
| 1024 | 6090 | 3550 | 94 | 0,0003184 | - | - | - |
| 2048 | 14220 | 7102 | 190 | 0,000334001 | - | - | - |
| 4096 | 32526 | 14206 | 382 | 0,000588 | - | - | - |
| 8192 | 73232 | 28414 | 766 | 0,0008501 | - | - | - |
| 16384 | 162834 | 56830 | 1534 | 0,0012531 | - | - | - |
| 32768 | 358420 | 113662 | 3070 | 0,00456469 | - | - | - |
| - | - | - | - | - | 0,0056893702 | -0,002139839 | -0,002139839 |

Tabela 1. Dados para calibragem dos coeficientes a, b e c.

| Tam. Vetor | Tam. Chave | Tam. Registi | Config. Vetor | limCusto | limParticao | limQuebras | Tempo Execução |
|---|---|---|---|---|---|---|---|
| 2000 | 4 | 4 | inversamente ordenado | 20 | 1199 | 2 | 0,1376647 |
| 2000 | 4 | 4 | 25% ordenado | 20 | 27 | 12 | 0,1023472 |
| 2000 | 4 | 4 | 75% ordenado | 20 | 65 | 9 | 0,07882446 |
| 2000 | 4 | 4 | ordenado | 20 | 1199 | 2 | 0,0776174 |
| 2000 | 4 | 4 | 1011 quebras | 20 | 12 | 13 | 0,096722 |
| 2000 | 4 | 8 | 1011 quebras | 20 | 12 | 13 | 0,1158718 |
| 2000 | 4 | 16 | 1011 quebras | 20 | 12 | 13 | 0,13315304 |
| 2000 | 4 | 32 | 1011 quebras | 20 | 12 | 13 | 0,13809936 |
| 2000 | 2 | 8 | 1011 quebras | 20 | 12 | 13 | 0,1105872 |
| 2000 | 4 | 8 | 1011 quebras | 20 | 12 | 13 | 0,112445 |
| 2000 | 8 | 8 | 1011 quebras | 20 | 12 | 13 | 0,1134344000 |

Tabela 2. Dados obtidos variando os parâmetros antes fixos.

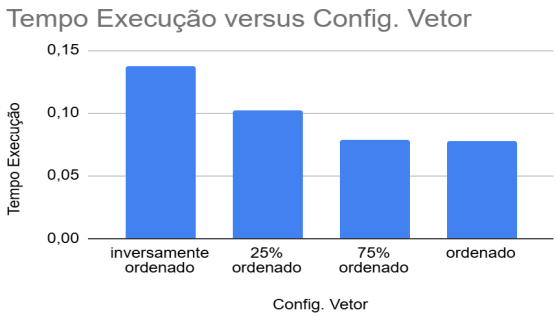


Gráfico 1. Relação entre o tempo de execução e a configuração do vetor de entrada.

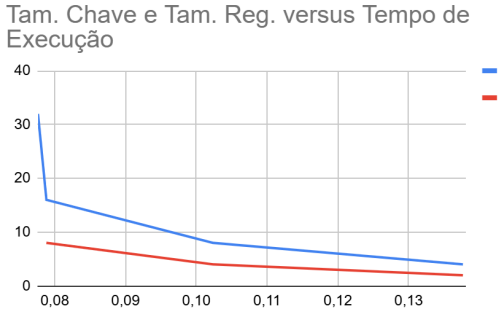


Gráfico 2. Relação entre o tamanho da chave (em vermelho) e o tempo de execução; e entre o tamanho do registro (em azul) e o tempo de execução.

Por fim, foram comparados os desempenhos do TAD Ordenador Universal (utilizando os limiares calibrados) e do algoritmo Quick Sort otimizado com mediana de 3 e Insertion para partições menores que 50. Para os testes, foram utilizados os exemplos de entradas fornecidos para o trabalho. Os resultados obtidos seguem na Tabela 3 e nos Gráficos 3 e 4.

| teste | tam. vet. | Quebras | cmp QS | cmp OU | moves QS | moves OU | calls QS | calls OU | tempo OU (s) | tempo QS (s) | custo QS | custo OU |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1500 | 758 | 19085 | 18594 | 12978 | 15545 | 2720 | 319 | 0,00029 | 0,0004082 | 196,2156 | 132,398068 |
| 2 | 1500 | 745 | 18855 | 18483 | 12885 | 15454 | 2678 | 343 | 0,0003306 | 0,00043626 | 193,322667 | 132,044103 |
| 3 | 1500 | 745 | 18855 | 23252 | 12885 | 21123 | 2678 | 159 | 0,0003244 | 0,0004246 | 193,322667 | 150,67788 |
| 4 | 1500 | 745 | 18855 | 23252 | 12885 | 21123 | 2678 | 159 | 0,0003082 | 0,0003988000 | 196,88262 | 155,072508 |
| 5 | 1500 | 747 | 18967 | 23287 | 12870 | 21242 | 2696 | 142 | 0,00046 | 0,000458 | 206,763786 | 154,877676 |
| 6 | 1500 | 747 | 18967 | 23287 | 12870 | 21242 | 2696 | 142 | 0,0004766 | 0,0014446 | 206,763786 | 154,877676 |
| 7 | 700 | 352 | 8059 | 7716 | 5427 | 6307 | 1258 | 184 | 0,0001872 | 0,0002628 | 90,399392 | 58,761279 |
| 8 | 200 | 93 | 1965 | 2000 | 1251 | 1737 | 358 | 34 | 0,0000546 | 0,0001402 | 23,54276 | 14,301264 |
| 9 | 2000 | 1011 | 28212 | 27596 | 17577 | 20923 | 3602 | 442 | 0,0004236 | 0,0006486 | 309,254533 | 216,193773 |
| 10 | 50 | 25 | 369 | 285 | 243 | 249 | 90 | 22 | 0,0000408 | 0,0000786 | 4,831524 | 2,376576 |

Tabela 3. Resultados dos testes comparativos entre QuickSort e Ordenador Universal.

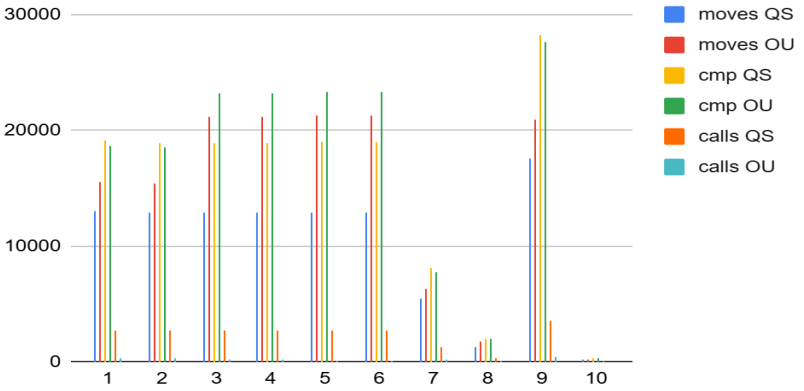


Gráfico 3. Resultados comparativos em termos de estatísticas de comparações, movimentações e chamadas de função.

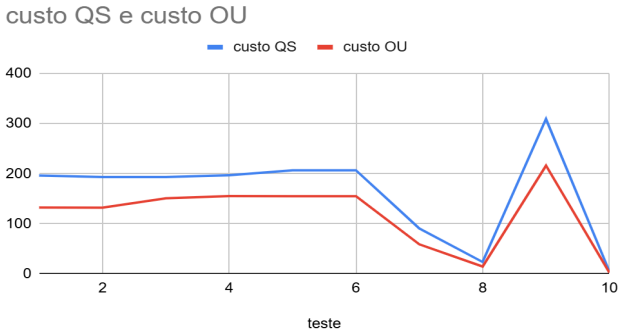


Gráfico 4. Resultados comparativos em termos de custo de execução para Quick Sort (QS) e Ordenador Universal (OU).

### 5.2 Conclusões

Na segunda parte da análise experimental, como esperado, nota-se que o tempo de execução do algoritmo cresce à medida que o nível de desordenação da entrada aumenta. Além disso, tem-se que os limiares de quebras e partição permanecem constantes nos extremos da faixa de testes e variam para valores intermediários de ordenação, evidenciando o impacto direto da organização inicial para o funcionamento do TAD.

Em seguida, observa-se que o tamanho do registro é o fator que mais gera aumentos no tempo de execução quando variado, com exceção do caso de entrada ordenada e inversamente ordenada, que gera a maior diferença de tempo observada. Este resultado é coerente já que valores maiores de registro geram maior gasto computacional para movimentações, o que gera o aumento do tempo de execução. A variação do tamanho da chave também gera aumento no tempo de execução, mas com alterações mais discretas. Este resultado também é esperado, já que operações de comparação (que se tornam mais caras à medida que a chave cresce) não são tão custosas quando as de movimentações de registro.

Por fim, a análise comparativa de algoritmos destaca a eficiência do TAD Ordenador Universal: para diversos tamanhos de entrada, coeficientes de custo e organizações do vetor de entrada, a ordenação do TAD é sempre menos custosa e pelo menos tão rápida quanto a do Quick Sort, que já é um método considerado ótimo. É interessante notar que o melhor desempenho do TAD não necessariamente ocorre porque as suas estatísticas são sempre menores que as do Quick Sort. Muitas vezes o método apresenta número de comparações e movimentações maior do que o próprio Quick Sort, por exemplo, sendo que a sua maior otimização ocorre para o número de calls, que é sempre consideravelmente menor em

comparação. Ainda assim, ao fim do cálculo de custo, o TAD obtém resultados melhores e tempos de execução menores, o que comprova sua adaptabilidade e superioridade prática frente a métodos clássicos como o Quick Sort.

## 6. Conclusões

## 7. Bibliografia