JOSÉ AUGUSTO SOARES PRADO

ANÁLISE EXPERIMENTAL DO *QUICKSORT* PROBABILÍSTICO COM GERADOR DE NÚMEROS PSEUDO-ALEATÓRIOS PENTA-INDEPENDENTE

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.
Orientador: Dr. Jair Donadelli Jr.

CURITIBA

2005

Sumário

Lista de Figuras

Glossário

- GNPA (RNG) - Gerador de Números Pseudo-Aleatórios (Random Number Genarator)
- Árvore do *Quicksort*- Árvore imaginária gerada pelos particionamentos do *Quicksort*
- QS-5i - *Quicksort* que usa GNPA Penta Independente na escolha do pivô
- QS-CL - *Quicksort* que usa GNPA por Congruência Linear na escolha do pivô
- QS-GNU - *Quicksort* versão implementada pela GNU
- QS-med5 - *Quicksort* que usa mediana de 5 na escolha do pivô
- QS-Fixo - *Quicksort* que usa pivô fixo na primeira posição da seqüência

Resumo

Estudamos, neste trabalho, diversas variações do *Quicksort* com relação à otimização do *cache* e percebemos que tal preocupação limita muito o algoritmo e as diferenças resultantes são extremamente específicas com relação à arquitetura utilizada ou simulada. Estudamos então variações do *Quicksort* que têm como foco modificar a forma de escolha do pivô com o objetivo de minimizar o número de comparações e trocas. Dentre as variações estudadas, os *Quicksort* probabilísticos caracterizam-se por usarem números pseudo-aleatórios na escolha do pivô. Concentramos-nos então em estudar dois algoritmos geradores de números pseudo-aleatórios (GNPA), um por congruência linear, proposto por Knuth em [Knu00b], e outro penta-independente proposto por Howard Karloff em [KR93]. Analisamos, comparamos e testamos esses dois GNPAs sendo usados na escolha do pivô do *Quicksort*. Também comparamos essas duas versões probabilísticas com outras três implementações determinísticas que fizemos para o *Quicksort*. O intuito desse trabalho foi realizar uma análise experimental dessas variações do *Quicksort* usando GNPAs dando enfoque em analisar o comportamento da versão proposta em [KR93].

Abstract

We had studied in this work variations of *Quicksort* related with cache optimization and we had realized that worry about this topic makes the algorithm deeply limited and the resultant diferences are extremely specific and related with the choosen architeture. So we had studied variations of *Quicksort* which focus in modifying the way of choosing the pivot with the objective of minimize the number of performed comparisions and changes. Among the looked variations, the probabilistic *Quicksort* caracterizes itself by using pseudo-random-numbers when choosing the pivot. So we had focused on study two random number genarator (RNG) algorithms, one by linear congruence [Knu00b] and another by five-way-independent [KR93]. We had analised, compared and tested these two RNGs beeing used on the pivot choice in *Quicksort*. We had also compared these two probabilistic versions with other three deterministic versions we had made for *Quicksort*. The aim of this work was to perform an experimental analysis of these variations of *Quicksort* using RNGs and the main focus was to analise the behavior of the version proposed in [KR93].

Capítulo 1

Introdução

O *Quicksort* foi inventado em 1962 por Hoare e é um algoritmo bom por ser relativamente fácil de ser implementado e realiza ordenação *in-place*, ou seja, não precisa de um vetor auxiliar para realizar a ordenação. O tempo médio de execução é ótimo $O(n \log n)$ onde n é o número de elementos na seqüência de entrada, porém sua implementação é frágil e deve ser realizada com cuidado para evitar a ocorrência do pior caso que é $O(n^2)$. Desde sua invenção, o *Quicksort* é provavelmente um dos mais utilizados algoritmos de ordenação até hoje e muitos pesquisadores tentam ainda melhorar a implementação do *Quicksort* propondo diferentes versões otimizadas desse algoritmo. Cada otimização leva em consideração um critério, uma idéia de melhoria, mas o princípio básico do *Quicksort* é sempre o mesmo, dividir para conquistar. Neste trabalho apresentaremos estudos feitos sobre diferentes idéias e diversas formas de se melhorar o *Quicksort* para casos específicos. Mostraremos, também neste trabalho, uma análise experimental realizada sobre cinco diferentes implementações do *Quicksort* com ênfase nas probabilísticas.

1.1 Análise Experimental de Algoritmos

É uma área de estudo que vem sendo aperfeiçoada por diversos pesquisadores. O objetivo de se realizar experimentação com algoritmos é verificar se o comportamento previsto na análise teórica corresponde à realidade de um processo sendo executado sobre um sistema real. Definir a plataforma de testes (comparabilidade) e realizar diversos testes para cada caso (confiabilidade) são cuidados que precisam ser tomados em uma análise experimental. Quando queremos conhecer o comportamento de um algoritmo não é necessário muito esforço para encontrar motivações que nos levem a usar experimentação. Os modelos assintóticos de pior caso são muito pessimistas, modelos realísticos são muito complicados, modelos padrão podem ser muito simples e a maioria dos algoritmos são bastante complicados para que possamos construir todas as ferramentas assintóticas analíticas necessárias para simular o ambiente real. Em [Nom03] a análise experimental foi amplamente usada para aperfeiçoar o algoritmo foco do trabalho (algoritmo para teste de planaridade de grafos), com os resultados obtidos na experimentação de uma versão, o autor, combinando esses resultados com estudos, produzia a próxima versão. McGeoch em [McG86a] defende a experimentação e mostra diversos pontos interessantes a serem considerados para quem quer realizar uma boa análise experimental de algoritmos. Ela comenta também que dificuldades associadas com precisão numérica, não aleatoriedade de números pseudo-aleatórios em Geradores de Números Pseudo-Aleatórios (GNPA) e ferramentas de medição não precisas podem gerar resultados inesperados e/ou errados.

1.2 Algoritmos Probabilísticos

Um algoritmo probabilístico é um algoritmo que recebe como entrada, além da instância do problema, uma seqüência de bits aleatórios que serão usados em algum processo decisório no decorrer do algoritmo. Sendo assim, para uma mesma instância, o algoritmo pode gerar resultados diferentes pois tomará diferentes decisões em seu percurso. Na experimentação com algoritmos probabilísticos deve-se calcular a média de seus resultados para uma mesma entrada padrão, e comparar o seu desempenho com o de algoritmos determinísticos conhecidos para o mesmo problema. A análise do caso médio do *Quicksort*, por exemplo, depende de que a entrada seja aleatóriamente ordenada, ou seja, uma distribuição uniforme. Em um algoritmo probabilístico

não devemos nos preocupar só com a entrada, pois a média de desempenho depende não só da entrada mas também dos números aleatórios ou pseudo-aleatórios usados durante o processo decisório. A análise assintótica de algoritmos probabilísticos normalmente leva em consideração que temos uma fonte inesgotável de números verdadeiramente aleatórios, mas queremos saber também o que acontece se usarmos números pseudo-aleatórios. Os algoritmos probabilísticos estão se tornando cada vez mais comuns e muitos já se mostraram melhores do que seus correspondentes determinísticos. Esse ramo de possibilidades de melhoria foi sem dúvida parte da inspiração para este trabalho.

1.3 O *QS-5i*

O *Quicksort* probabilístico padrão usa $O(n \log n)$ bits aleatórios pois usa um número aleatório a cada partição e roda em tempo esperado $O(n \log n)$. O *Quicksort* probabilístico com gerador de congruência linear usa apenas $O(\log n)$ bits aleatórios mas requer tempo esperado $\Omega(n^4/m^2)$ onde n é o tamanho da entrada do *Quicksort* e m é o periodo dado pelo gerador por congruência linear [KR93]. O *Quicksort* com GNPA Penta-Independente (*QS-5i*) usa apenas $O(\log n)$ bits aleatórios e mantém o tempo esperado $O(n \log n)$. Neste trabalho mostramos essas previsões assintóticas, no entanto mostramos também que na média o *QS-5i* não supera significativamente o *Quicksort com GNPA por Congruência Linear*(QS-CL), e comparamos ambos com o *Quicksort* mediana-de-3 utilizado pela GNU no ambiente testado e também ao *Quicksort* mediana-de-5 que construímos neste trabalho.

1.4 Organização deste trabalho

No capítulo 2 mostramos como funciona o *Quicksort* tradicional. Também vemos várias alterações possíveis no *Quicksort* para que este se comporte de maneira melhor, ou seja, menor probabilidade de ocorrer o pior caso. Podemos observar nesse capítulo que o principal foco das otimizações é a escolha do pivô na função particiona, porém existem otimizações com outros pontos de vista. Na seção 2.3 mostramos a análise do caso médio do *Quicksort* determinístico para que tenhamos uma previsão assintótica do comportamento do *Quicksort*. Na seção 2.5 estudamos otimizações realizadas por outros autores a com foco no melhor aproveitamento do *cache*. Apesar de inte-

ressante, essas otimizações não são o foco deste trabalho. No capítulo 3 nos referenciamos à números pseudo-aleatórios e suas aplicações em algoritmos probabilísticos, mostramos um GNPA conhecido, por congruência linear [Knu00b], e ressaltamos um problema apresentado em [KR93]. Estudamos então a solução do *Quicksort* probabilístico com GNPA penta-independente (QS-5i). No capítulo 4 temos os dados da análise experimental que fizemos com foco principal de testar a desempenho do QS-5i. Percebemos que apesar do QS-5i apresentar justificativas assintóticas aceitáveis de que seria melhor do que o QS-CL, na média o QS-5i se apresentou pior do que o QS-CL. Mas este fato a primeira vista curioso é justificável pois a superioridade assintótica do QS-5i se da apenas para o pior caso e não para o caso médio que é o que tende a ocorrer em uma experimentação. No capítulo 5 apresentamos a conclusão. O apendice A contém os principais dados numéricos e médias obtidos durante os mais de 6000 testes realizados no decorrer da fase de testes experimentais deste trabalho. Os códigos criados neste trabalho podem ser encontrados em `http://web.inf.ufpr.br/josea` e as entradas aleatórias utilizadas foram providas pelo grupo random.org em `http://www.random.org`.

Capítulo 2

Quicksort e variações

O algoritmo *Quicksort* foi proposto em 1962 por C. A. R. Hoare e, desde então, vem sendo estudado por muitos pesquisadores. Muitas otimizações para casos particulares já são conhecidas.

O *Quicksort* é um algoritmo do tipo dividir para conquistar para ordenação cujo tempo de execução no pior caso é $O(n^2)$ sobre uma seqüência de entrada de n números. Apesar desse tempo de pior caso ser ruim, esse algoritmo se destaca entre os algoritmos de ordenação porque seu tempo médio é muito mais próximo do melhor caso do que do pior caso.

O custo por nível de recursão é de aproximadamente n comparações e são $\log n$ níveis no melhor caso, e no pior caso $n - 1$ níveis. Vamos considerar que o *Quicksort* terá sempre como entrada uma seqüência numérica a ser ordenada.

Neste capítulo apresentamos uma revisão bibliográfica das variantes do *Quicksort* encontradas na literatura, assim como a avaliação do desempenho dessas variantes. Na seção 2.7 mostramos estudos que levam em conta efeitos de memória *cache* no desempenho.

2.1 O algoritmo

O algoritmo *Quicksort* pode ser escrito da seguinte maneira:

Algoritmo 1 Quicksort(*sequencia*, *inicio*, fim)

```
se fim − inicio ≤ 1
  retorna
} fim se
pivo ← particiona(sequencia, inicio, fim)
Quicksort (sequencia , inicio , pivo − 1 );
Quicksort (sequencia , pivo + 1 , fim );
```

O elemento mais importante do *Quicksort* é a função `particiona`, quanto mais otimizada for essa função, mais rápida será a ordenação. Otimizações da função `particiona` não alterarão a escala de complexidade do algoritmo mas podem causar melhorias relevantes no tempo médio de resposta quando implementadas. A maioria das variações do *Quicksort* são principalmente variações da função `particiona`. Na próxima seção apresentamos essa função.

2.1.1 Particionamento

A função `particiona` elege um dos elementos da seqüência como elemento **pivô**. Após essa eleição, cada elemento da seqüência é comparado com o pivô e, se for menor que o pivô, deverá ficar à esquerda desse, se for maior que o pivô, deverá ficar à direita. Ao final dessa etapa, o elemento pivô fica na sua posição final da seqüência ordenada.

Uma forma de se implementar o particionamento é fazer com que dois marcadores se desloquem, um da direita para a esquerda e o outro da esquerda para a direita, comparando os elementos com o pivô até se encontrarem no meio.

No algoritmo 2, consideramos que a função `elegepivo` é uma função que escolhe o pivô por algum critério previamente definido; várias opções para essa função serão mostradas no decorrer deste trabalho.

O tempo (número de comparações) para a execução da função `particiona` é $n + 1$.

Algoritmo 2 Particiona(*sequencia*, *inicio*, fim)

```
i ← inicio
j ← fim
indicePivo ← elegepivo()
pivo ← sequencia[indicePivo]
enquanto 1
  enquanto sequencia[i] < pivo
    i ← i + 1
    se i = fim
      break
    } fim se
  } fim enquanto{posicionou o i no primeiro elemento da esquerda para a direita que é maior
  (ou igual) ao pivô}
  enquanto pivo < sequencia[j]
    j ← j − 1
    se j = inicio
      break
    } fim se
  } fim enquanto{posicionou o j no primeiro elemento da direita para a esquerda que é menor
  (ou igual) ao pivô}
  se i ≥ j
    break
  } fim se{se os ponteiros se cruzarem então pare}
  troca(sequencia[i], sequencia[j]) {faz a troca, colocando elementos menores ou iguais ao pivo
  para a esquerda, e elementos maiores ou iguais ao pivô para a direita}
} fim enquanto
troca(sequencia[i], sequencia[indicePivo])
retorna i {porque i é o novo índice do pivo agora ja na posição correta}
```

2.1.2 A escolha do pivô é que faz a grande diferença

A escolha do pivô determina o quão balanceado será aquele particionamento. Um particionamento é otimamente balanceado quando o elemento escolhido como pivô é o elemento central da seqüência ordenada, ou seja, a mediana de todos os elementos. Um particionamento é pessimamente balanceado quando o elemento escolhido como pivô é o maior ou o menor elemento da seqüência.

O melhor caso do *Quicksort* ocorre quando todos os particionamentos em todos os níveis de recursão forem ótimos, ou seja, quando toda vez que a função `particiona` for chamada, ela divida a seqüência exatamente ao meio. Isso pode ser visto como uma árvore binária (Figura 2.1) onde a profundidade de todas as folhas é a mesma ($\lceil \log_2 n \rceil$ no caso). Nesse exemplo $T(n) = T(n/2) +$

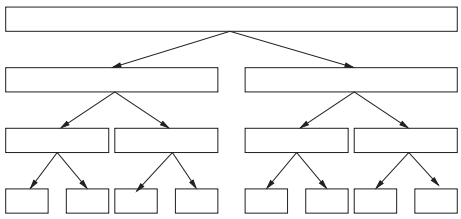

Figura 2.1: Melhor caso.

$\Theta(n) = \Theta(n \log n)$ onde $T(n)$ é o número de comparações para uma entrada com n elementos. Num certo sentido, o pior caso do *Quicksort* ocorre quando todos os particionamentos de todos os níveis de recursão ocorrem da pior maneira possível, ou seja, o elemento escolhido como pivô pela função `particiona` for sempre o maior ou o menor elemento da seqüência. Isso pode ser visto como uma árvore binária com uma folha por nível (Figura 2.2) que mais se parece com uma lista e tem profundidade $n - 1$. Nesse caso, $T(n) = T(n-1) + \Theta(n) = \Theta(n^2)$. Tomando o caso

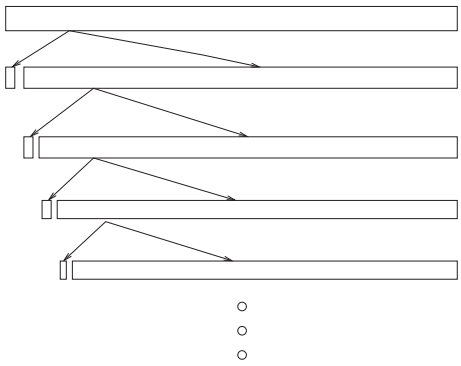

Figura 2.2: Pior caso.

balanceado (α ,β) do *Quicksort* temos $T(n) = T(\alpha n) + T(\beta n) + \Theta(n)$ com $0 < \alpha \leq 1/2$ e $\alpha + \beta = 1$ constantes. O ramo mais curto na árvore tem altura $O(\log_{1/\alpha} n)$ e o mais longo $O(\log_{1/\beta} n)$. O número de comparações é $\Theta(n \log n)$. No pior caso temos $T(n) = \max(T(q) + T(n-q)) + O(n)$ e segue por indução que $T(n) \leq cn^2$ para alguma constante $c > 0$ e todo n suficientemente grande.

2.2 Variações do algoritmo

O *Quicksort* é um algoritmo de ordenação muito bom na média, porém algumas entradas podem causar desempenho ruim e por isso ao longo dos anos foram desenvolvidos algumas técnicas a serem utilizadas na implementação para evitar que tempos da ordem de n^2 aconteçam.

2.2.1 Eliminar recursões em pequenas subseqüências

O *Quicksort* funciona muito bem para seqüências grandes, porém não é muito eficiente para seqüências de tamanho pequeno. Isso ocorre porque para seqüências pequenas o tempo do *Quicksort* será quadrático também e ocupará mais espaço em memória devido aos particionamentos.

No caso de seqüências pequenas é melhor, com relação ao tempo de resposta, usar um algoritmo como o *Insertionsort* [SF96].

Levando esse pensamento adiante, podemos então mudar o teste que termina a recursão do *Quicksort* de forma que, ao invés de parar quando a subseqüência tiver tamanho 1, pare quando ela tiver tamanho M e utilize outro método de ordenação.

Ao invés de termos as linhas:

se $(fim - inicio \leq 1)$ **então**
  retorna {condição de término da recursão}
fim se

no algoritmo, teríamos:

se $(fim - inicio \leq M)$ **então**
  InsertionSort(*sequencia*,*inicio*,fim); {condição de término da recursão}
fim se

O valor exato de M vai depender da implementação utilizada, [Sed78] sugeriu valores entre e 6 e 15.

Sedgewick propõe em [Sed78] que o *Insertionsort* não seja chamado como mostrado acima, mas sim que o particionamento ignore as pequenas subseqüências. Ao final do processo, sobra uma seqüência quase ordenada com elementos fora de ordem entre si apenas em intervalos de tamanho M. O *Insertionsort* então é chamado apenas uma vez para esta seqüência quase ordenada.

[Sed78] mostrou que essa abordagem é melhor do que chamarmos o *Insertionsort* toda vez que encontrarmos uma pequena subseqüência.

Essa idéia foi descartada por LaMarca em [LL97], veja 2.7.1, no seu *Memory-Tunned-Quicksort*, ele levou em consideração o ponto de vista de otimizar a utilização do *cache*, e mostrou que é melhor chamar diretamente o *Insertionsort* quando as pequenas subseqüências forem encontradas pois essas acabaram de passar por um particionamento e, portanto, ainda devem estar no *cache*. Desse modo, o tempo real de execução do *Insertionsort* é menor.

2.2.2 Particionamento aleatório e mediana-de-três

Como já vimos, o principal fator na determinação do desempenho de uma execução do *Quicksort* é a escolha do pivô. Existem então alguns métodos de escolha que podem melhorar o desempenho do *Quicksort* na média de muitas execuções, pois evita que o pior caso aconteça e tenta tendenciar a escolha do pivô para mais próximo do centro da seqüência ordenada (o que seria o melhor caso).

Uma forma de melhorar a escolha do pivô é usar um gerador de números aleatórios para escolher o pivô, que descaracteriza o pior caso. Em aplicações comerciais é comum que processos automatizados tentem ordenar uma seqüência já ordenada, sendo assim, o pior caso do *Quicksort* ocorre com uma determinada freqüência; usando-se um gerador de números aleatórios para escolher o pivô, essa probabilidade diminui.

Considere que γ é a quantidade de elementos de cada subseqüência de recursão ou iteração.

A probabilidade de ocorrer o pior caso é $2/\gamma$ para cada subseqüência. O porquê de $2/\gamma$ se explica pois o pior caso pode ocorrer escolhendo-se o menor ou o maior elemento da seqüência. Sendo assim, como temos $n-1$ níveis de recursão no pior caso e, lembrando que a probabilidade de um evento ocorrer repetidas vezes, independentemente, é o produto da probabilidade desse evento ocorrer em casos isolados, temos probabilidade

$$\left(\frac{2}{n}\right) \cdot \left(\frac{2}{n-1}\right) \cdot \left(\frac{2}{n-2}\right) \cdot \left(\frac{2}{n-3}\right) \cdot \left(\frac{2}{n-4}\right) \cdot \ldots \cdot \left(\frac{2}{2}\right)$$

de ocorrer o pior caso. Organizando melhor, temos então probabilidade

$$\prod_{x=0}^{n-2} \frac{2}{(n-x)} = \frac{2^{n-1}}{n!} = o(1)$$

de ocorrer o pior caso.

Uma possível alternativa para diminuir a probabilidade do pior caso é escolher o pivô como mediana de uma amostra.

O particionamento mediana-de-três consiste em selecionar três elementos como pré-pivôs (amostra), sejam estes por exemplo, os três da esquerda, os três da direita ou os três do centro. Seleciona-se então a mediana desses três elementos e então essa é eleita pivô. Isso dimunui a probabilidade de que o pivô seja o maior ou o menor elemento da seqüência.

2.2.3 Particionamento mediana-de-*T*

Consideremos que T é o tamanho de uma amostra de uma subseqüência, os elementos dessa amostra são os pré-pivôs e a mediana dos pré-pivôs será o pivô usado no particionamento. Na seção anterior, por exemplo, onde vimos mediana-de-três, significa que T é igual a três. McGeoch estudou a possibilidade de variar T.

McGeoch mostrou em [McG86b] que a estratégia de se fazer escolha do pivô a partir da mediana de uma pequena amostra de elementos é uma boa estratégia. O desafio de escolher o T correto em função do tamanho da subseqüência foi o que inspirou McGeoch e Tygar [MT95] a realizarem um estudo mais detalhado sobre isso.

Nesse estudo, McGeoch e Tygar chamaram de estratégias *two-tier* (TT) aquelas que variam o tamanho T no primeiro estágio apenas, nos outros estágios de recursão utilizam um T fixo. Mostraram que a estratégia *two-tier* ótima para uma seqüência de comprimento n tem sempre um custo (número de comparações) menor do que qualquer T fixo para o mesmo n. Consideraram estratégias *square-root* (SQ) aquelas que usam a raiz quadrada de n como tamanho T da amostra, e fazem isso em todos os níveis de recursão. Provaram que a estratégia SQ (com $T = \theta(\sqrt(n))$ para cada nível de recursão) tem custo total menor do que qualquer estratégia TT para o mesmo n e também mostraram que o custo do pior caso para qualquer *Quicksort* SQ é $O(n^{1.5})$ [MT95] pág. 288.

Concluíram então que tanto a estratégia *square-root* quanto a *two-tier* requerem um número de comparações menor do que a estratégia de T fixo (escolha do pivô pela mediana de uma amostra de tamanho T).

Podemos genericamente dizer que neste caso (SQ) existem dois passos a mais em relação ao algoritmo original do *Quicksort*:

- Calcular T em função de n; o objetivo desse passo é achar o tamanho ideal da amostra (T) para um determinado n. Vamos chamar o custo total dessa operação de A.
- Determinar a mediana dos T elementos; o objetivo desse passo é calcular a mediana dos T elementos e achar o índice do pivô. Vamos chamar o custo total dessa operação de B.

Esses dois passos têm como objetivo equilibrar um pouco mais o particionamento do *Quicksort*. Isso acarreta em diminuição do número de recursões necessárias para concluir a ordenação, consequentemente diminuindo também o número total de comparações. Mas esse ganho (níveis de recursão ou iterações a menos) tem um custo ($A + B$), que obviamente não pode ser maior do que o ganho.

Em [MT95], foi concluído que o *Quicksort* (TT) e o *Quicksort* (SQ) eram melhores do que o *Quicksort* de Hoare que usa mediana de 3. Porém essa conclusão pode ter sido distorcida pelo fato de que só fora considerado o custo A. Martinez e Roura em [MR01] basearam seu estudo em [MT95] e consideraram tanto o custo A quanto o custo B. Eles mostraram então uma função para calcular T em função de duas variáveis, γ e a; onde γ é igual ao tamanho da seqüência em um dado nível de recursão atual, e a é uma constante que pode ser escolhida de acordo com o custo do algoritmo que será usado para calcular a mediana da amostra de tamanho T.

Por exemplo, usando o *QuickFind* de Hoare ([Knu00a], [CLRS02] cap 10) e não levando em conta o número de trocas, só o de comparações o número ótimo para tamanho da amostra T obtido em [MR01] é

$$T = \sqrt{\frac{1}{2\ln 2} \cdot \frac{1}{2+\ln 4} \cdot n} + o(\sqrt{n})$$

Os autores em [MR01] ainda conjecturaram que o tempo correto para $o(\sqrt{n})$ é $\Theta(1)$. Em [MR01] pode ser encontrado um resultado mais geral, porém, levando em conta o número de trocas.

Quanto mais custoso for o algoritmo para encontrar a mediana, menor será a; e conseqüentemente menor será T já que a função ótima determinada em [MR01] é $T = a \cdot \sqrt{\gamma} + o(\sqrt{\gamma})$.

Comentários sobre uma experimentação do algoritmo mediana-de-T podem ser encontrados na seção 2.5.

2.2.4 Particionamento triplo

Para seqüências com valores duplicados, o *Quicksort* se comporta bem, porém pode ser melhorado. Por exemplo, se tivermos uma seqüência inteira com o mesmo número, o *Quicksort* tradicional continuará dividindo a seqüência em subseqüências até completar toda a ordenação.

A idéia do particionamento triplo é de dividir a seqüência em três partes durante o particionamento, sendo a primeira com elementos menores que o pivô, a segunda com elementos iguais ao pivô e a terceira com elementos maiores do que o pivô, nessa ordem.

Esse particionamento é mais custoso do que o particionamento comum em duas partes, pois esse requer ao menos duas comparações por elemento por nível de recursão. Ou seja, para cada elemento deve-se verificar se ele é maior que o pivô, se não for, verifica-se se é igual, se não for então é porque é menor. Isso resulta em $2n$ comparações por nível de recursão, o que dobra o número de comparações do algoritmo tradicional. No entanto, essa abordagem pode reduzir muito o número de níveis de recursão, dependendo de quantos números iguais existam na seqüência de entrada, reduzindo então também o número de comparações.

2.3 Análise do caso médio do *Quicksort* determinístico

A seguir apresentamos uma análise do *Quicksort* determinístico com pivô fixo em qualquer posição. Consideramos que a seqüência de entrada é uma seqüência aleatória, onde os termos foram escolhidos com uma distribuição uniforme de probabilidades e não contém repetições. C_n é o número médio de comparações do *Quicksort* sobre uma entrada de tamanho n.

Podemos dizer que a probabilidade da partição ocorrer no k-ésimo elemento da seqüência ordenada é $1/n$ e que o número de comparações é

$$n + 1 + C_{k-1} + C_{n-k}.$$

Temos então

$$C_n = \frac{1}{n} \cdot \left(\sum_{k=1}^{n} (n + 1 + C_{k-1} + C_{n-k}) \right) = (n + 1) + \frac{2}{n} \left(\sum_{j=0}^{n-1} C_j \right)$$

para $n > 0$ e $C_0 = 0$. Podemos quebrar o somatório de somas em somas de somatório.

$$C_n = \left(\sum_{k=1}^{n} (n + 1) + \sum_{k=1}^{n} C_{k-1} + \sum_{k=1}^{n} C_{n-k} \right) / n$$

O primeiro termo está somando $(n + 1)$ durante n vezes e depois dividindo por n, então, isolando esse primeiro termo temos apenas $n + 1$. Extendendo a soma do segundo termo temos: $C_0 + C_1 + C_2 + C_3 + ... + C_{n-1}$ Extendendo a soma do terceiro termo temos: $C_{n-1} + C_{n-2} + C_{n-3} + ... + C_0$ ou seja, os dois são a mesma soma; então podemos dizer que a recorrência do caso médio do *Quicksort* Determinístico é:

$$C_n = (n + 1) + \frac{2}{n} \left(\sum_{j=0}^{n-1} C_j \right) \tag{2.1}$$

para $n > 0$; e $C_0 = 0$.

A Figura 2.3 mostra o número de comparações para o pior caso, para o caso médio e para o melhor caso, num intervalo onde n varia de 2 até 10.000. O eixo x é o valor de n, enquanto o eixo y é o número de comparações necessárias para a ordenação completa. É fácil perceber como o caso médio dado pela recorrência acima fica claramente mais próximo do melhor caso que é $n \lg n$ do que do pior caso que é n^2.

Teorema 1. *O número médio de comparações realizadas pelo Quicksort para uma permutação aleatória é* $\Theta(n \log n)$.

Demonstração.

$$C_n = 2(n + 1)(H_{n+1} - 1),$$

onde $H_n = \sum_{i=1}^{n} (\frac{1}{i})$ é o n-ésimo número harmônico. Sabemos que ([KGP94])

$$\ln(n) < H_n < \ln(n) + 1.$$

$$2n \ln(n) + 2 \ln(n) - 2n < C_n < 2n \ln(n) + 2 \ln(n) + 2,$$

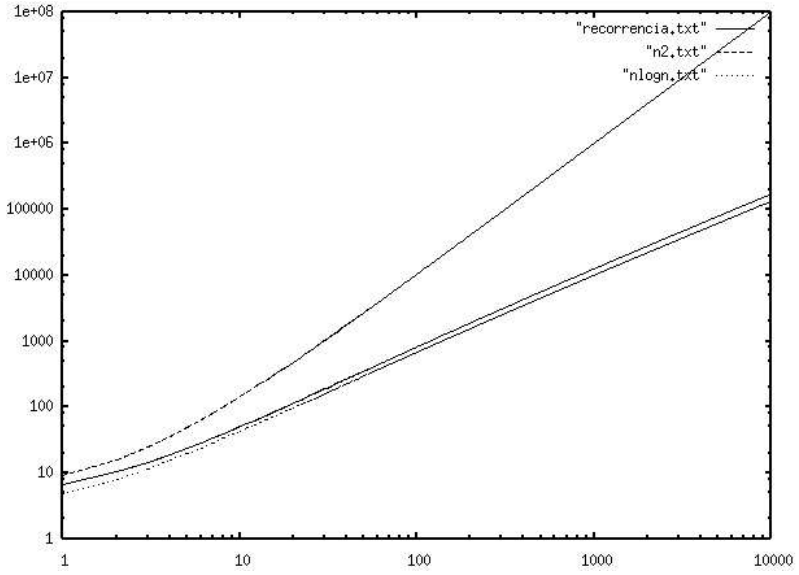

Figura 2.3: Comparação do caso médio com os casos extremos.

$$C_n = \Theta(n \log(n)) .$$

□

Teorema 2 ([SF96], pág. 111). *O número médio de comparações usadas pelo Quicksort mediana-de-3 para uma permutação aleatória (distribuição uniforme) é* $C_n = \frac{12}{7}(n+1)\left(H_{n+1} - \frac{23}{14}\right)$

Teorema 3 ([SF96]). *A recorrência do Quicksort de Sedgewick é*

$$C_n = \begin{cases} n+1+\frac{1}{n}\sum_{1\leq i\leq n}(C_{i-1}+C_{n-i}) & \text{para } n > M; \\ \frac{1}{4}n(n-1) & \text{para } n \leq M; \end{cases}$$

onde C_n *é o número de comparações ocorridas no algoritmo.*

Teorema 4 ([SF96], pág. 112). *O número médio de comparações usadas pelo Quicksort mediana-de-(2t+1) para uma permutação aleatória (distribuição uniforme) é* $\frac{1}{H_{2t+2}-H_{t+1}} n \ln n + O(n)$.

2.4 Versão probabilística com particionamento triplo

Vamos mostrar uma versão probabilística do *Quicksort* que tem tempo de execução esperado $O(n \log n)$ com alta probabilidade (veja o algoritmo 3). Existem provas mais simples para demonstrar que o tempo de execução esperado é $O(n \log n)$, mas essas provas não servem para mostrar que isso acontece com alta probabilidade.

Algoritmo 3 QuickSort Probabilístico(seqüência S)

`se` $S \leq 20$
  ordene S com *Insertionsort* e devolva S
} `fim se`
Escolha aleatoriamente um elemento M(S) como pivo

Compare cada elemento S com o pivo e particione S em três subseqüências S_1 S_2 S_3 onde S_1 contém os elementos de S menores que o pivo, S_2 contém os elementos de S iguais ao pivo e S_3 contém os elementos de S maiores que o pivo.

Ordene S_1 e S_3 recursivamente.

Devolva S_1 concatenada com S_2 concatenada com S_3.

Seja X a variável aleatória que representa o número de comparações entre elementos de S com um pivô, durante uma execução do *Quicksort* Probabilístico numa seqüência S de n elementos.

A distribuição de probabilidade aqui é sobre a escolha do pivô, diferentemente do caso médio onde a distribuição é sobre a entrada.

Teorema 5. $\mathrm{E}(X) = O(n \log n)$

Teorema 6. $X = O(n \log n)$ *acontece com probabilidade* $1 - O(n^{-6})$ *(alta probabilidade).*

Demonstração. $\mathrm{E}(X) = O(n \log n)$

Um elemento de S pode ser comparado com pivô em vários níveis de recursão.

Para simplificar a escrita das expressões, os logaritmos usados daqui até o fim da análise probabilística são na base $\frac{8}{7}$.

Seja $S = (e_1, e_2, \ldots, e_n)$ e seja X_i o número de vezes que o elemento e_i é comparado com o pivô.

Então, temos que $\mathrm{E}(X) = \sum_{i=1}^{n} \mathrm{E}(X_i)$. Queremos provar que $\mathrm{E}(X) = O(n \log n)$. Para isso vamos mostrar que $\mathrm{E}(X_i) = O(\log n)$ para cada i, $1 \leq i \leq n$.

Ao final da execução teremos passado por uma árvore de recursão onde cada grupo de no máximo 20 elementos de S é uma folha da árvore.

Se X_i é o número de comparações do elemento e_i com um pivô, então X_i é a profundidade em que o elemento e_i se encontra na árvore de recursão. Sendo assim, após o término do algoritmo, podemos dizer que X_i é a profundidade na árvore para uma determinada folha que contenha e_i (Figura 2.4).

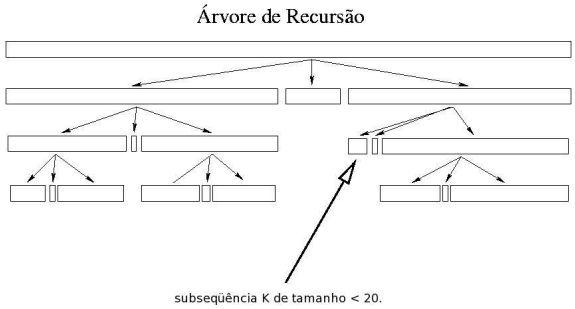

Figura 2.4: Árvore de recursão.

Por exemplo, para um elemento e_i que esteja na subseqüência K representada na Figura 2.4, $X_i = 3$. O que queremos calcular agora é o valor esperado para todos os X_i, ou seja, a profundidade média das folhas da árvore.

Vamos considerar então que um particionamento é *sucesso* se a menor parte resultante desse particionamento contém ao menos $\frac{1}{8}$ da seqüência anterior. Vamos considerar *insucesso* se o particionamento for menos balanceado que isso, ou seja, se uma das subseqüências for menor que $\frac{1}{8}$ da seqüência anterior.

Considerando que a probabilidade da partição ocorrer em qualquer lugar da seqüência é igual, então um *insucesso* acontece se for escolhido como pivô um dos $\frac{1}{8}$ menores ou um dos $\frac{1}{8}$ maiores elementos de S. Assim, a probabilidade de *sucesso* é de pelo menos $\frac{3}{4}$ e a probabilidade de *insucesso* é de no máximo $\frac{1}{4}$ para cada partição que ocorra. Cada elemento e_i participará em no máximo $\log(n/20)$ partições com *sucesso*.

Seja Y_k a variável aleatória que representa o número de partições entre uma partição com *sucesso* (exclusive) e a próxima partição com *sucesso* (inclusive), temos que a profundidade da

folha que contém e_i é $X_i = \sum_{k=1}^{\log(n/20)} Y_k$ e pela linearidade do valor esperado temos que

$$\mathrm{E}(X_i) = \sum_{k=1}^{\log(n/20)} \mathrm{E}(Y_k).$$

Se mostrarmos que $\mathrm{E}(Y_k)$ é uma constante, ou seja, independente de n, provaremos que $\mathrm{E}(X_i) = \mathrm{O}(\log n)$.

Sabendo que a probabilidade de *sucesso* de uma partição é de pelo menos $\frac{3}{4}$, logo o número esperado de *insucessos* antes que se consiga um *sucesso* e inclusive um *sucesso* é de no máximo $\frac{4}{3}$ pois Y_k tem distribuição geométrica [CLRS02].

Portanto $\mathrm{E}(Y_k) \leq \frac{4}{3}$ para todo k e, então $\mathrm{E}(X_i) = O(\log n)$ como queríamos provar. □

O número total de comparações no *Insertionsort* com uma seqüência de 20 elementos é

$$\binom{20}{2} = 190.$$

O número de comparações no *Insertionsort* que é chamado dentro do *Quicksort* Probabilístico de no máximo 190 para cada subseqüência folha, logo, no máximo $190(n/20)$ comparações, para o algoritmo todo, o que é $O(n)$.

Sendo assim, podemos dizer que a profundidade média da árvore de recursão do *Quicksort* Probabilístico é de $\left(\frac{4}{3}\right) \log n$; para cada nível de recursão temos $O(n)$ comparações. Então, o número esperado de comparações para a execução do algoritmo *Quicksort* Probabilístico é no máximo

$$\left(\frac{4}{3}\right) n \log n + O(n).$$

Demonstração. $X = O(n \log n)$ acontece com probabilidade $1 - O(n^{-6})$ (alta probabilidade).

Provamos então um limitante superior para o tempo esperado de execução do *Quicksort* Probabilístico. Porém, saber que um algoritmo é em média bom, não nos garante que ele é bom com alta probabilidade.

Seja A_i o evento $X_i > 8 \log n$. Lembremos que X_i é o número de comparações para o elemento

e_i, então se $X_i \leq 8\log n$ é porque temos um bom X_i. Provaremos que

$$Pr(X_i > 8\log) \leq n^{-7}.$$

A probabilidade de acontecer um evento ruim (A_i) é pequena. Se essa probabilidade é menor que n^{-7} então a probabilidade de acontecer algum evento ruim é

$$Pr\left(\bigcup_i A_i\right) \leq \sum_i Pr(A_i) \leq n \cdot n^{-7} = n^{-6}.$$

Lembremos que uma vez que a escolha dos pivôs é independente, os eventos de que as várias partições sejam *sucesso* também são eventos independentes. Qual a probabilidade de e_i passar por pelo menos $8\log n - \log(n/20)$ partições com *insucesso* em $8\log n$ partições, cada partição com probabilidade de *insucesso* de $\frac{1}{4}$? Seja Y o número de partições com *insucesso* em $8\log n$ partições, cada *insucesso* com probabilidade de no máximo $\frac{1}{4}$. Então

$$Pr(Y \geq 8\log n - \log(n/20)),$$

pois $8\log n - \log(n/20)$ é o número mínimo de partições com *insucesso*. Podemos simplificar os cálculos mantendo o que nos interessa, que é um número pequeno multiplicando $\log n$

$$Pr(Y \geq 8\log n - \log(n/20)) \leq Pr(Y > 7\log n).$$

A última expressão é menor ou igual a somatório da probabilidade dos eventos encadeados acontecerem, multiplicado por todas as combinações possíveis de *sucesso* e *insucesso* (j *insucesso* e $(8\log n) - j$ *sucessos*) para uma mesma quantidade de *insucesso* (j), multiplicado por todas as possibilidades de quantidades de *insucesso*:

$$Pr(Y > 7\log n) \leq \sum_{j>7\log n} \binom{8\log n}{j} \left(\frac{1}{4}\right)^j \left(\frac{3}{4}\right)^{8\log n - j} \leq \sum_{j>7\log n} \binom{8\log n}{j} \left(\frac{1}{4}\right)^j.$$

Usando a desigualdade $C_k^n \leq (\frac{ne}{k})^k$ temos então

$$\sum_{j>7\log n} \binom{8\log n}{j} \left(\frac{1}{4}\right)^j \leq \sum_{j>7\log n} \left(\frac{8e\log n}{j}\right)^j \left(\frac{1}{4}\right)^j = \sum_{j>7\log n} \left(\frac{8e\log n}{4j}\right)^j \leq$$

$$\sum_{j>7\log n} \left(\frac{2e\log n}{7\log n}\right)^j \leq \sum_{j>7\log n} \left(\frac{2e}{7}\right)^j \leq \sum_{j>7\log n} \left(\frac{7}{8}\right)^j.$$

A última expressão é uma série geométrica e soma $O(n^{-7})$.
Então $Pr(Y > 7\log n) \leq O(n^{-7})$ para cada e_i e A_i é igual ao evento $Y > 8\log n$. Somando todas as probabilidades de todos os e_i temos o que queríamos:

$$Pr\left(\bigcup_i A_i\right) \leq n^{-6}.$$

Então, podemos dizer que com probabilidade $1 - O(n^{-6})$ (alta probabilidade) X_i é menor do que $8\log n$, ou seja, o algoritmo *Quicksort* Probabilístico executa menos do que $8n\log n$ comparações.

□

2.5 Experimentação da mediana-de-*T* no *Quicksort*

Um método bem conhecido para escolher o pivô durante a partição do *Quicksort* é usar a mediana de uma amostra de tamanho T escolhida aleatoriamente a partir da seqüência original. Uma especialização desse método foi explicada na seção 2.2.2. A melhor escolha desse tamanho T está em algum ponto intermediário entre o custo de achar essa mediana (quando menor o T menor o custo) e uma melhor partição (quanto maior o T melhor é a partição).

Um detalhe interessante é que não foi executado e medido várias vezes o algoritmo do *Quicksort*, foi criado um algoritmo de simulação chamado de *shortcut* (algoritmo 4). Esse algoritmo simula o *Quicksort*, porém não ordena nada, apenas segue os passos do *Quicksort* incrementando variáveis para contar o número de instruções esperadas para as respectivas etapas do algoritmo.

Considere que M é o tamanho da subseqüência onde o *Quicksort* pára sua recursão e chama o *InsertionSort*, T é o tamanho da amostra a ser usada para achar o pivô e n é o tamanho da entrada

e sempre é uma potência de dois.

Algoritmo 4 Shortcut(n)

1: `se` $n < M$
2: $D+ = n - H_n$ {O número de *InsertionSorts* ocorridos}
3: $E+ = n(n-1)/4$ {O número total de movimentos durante os *InsertionSorts*}
4: `senão`
5: $T = t(n)$ {Determina o tamanho da amostra}
6: $M = (T+1)/2$
7: $pivo =$ geraPivo(n, T)
8: $A+ = 1$ {O número esperado de vezes que a rotina *Quicksort* é chamada}
9: $B+ = (n - pivo)(pivo - 1)/\binom{n}{T}$ {O número total esperado de trocas realizadas durante os particionamentos}
10: $C+ = n - 1$ {O número total de comparações realizadas durante os particionamentos (mas não durante a seleção da mediana)}
11: $F = 2((n+1)H_n - (M+1)H_M - (T-M+2)H_{T-M+1} + T + 5/3$ {C+F é o número total de comparações esperadas para o Fixed-T-QuickSort (*Quicksort* com T fixo em todos os níveis)}
12: Shortcut$(pivo - 1)$
13: Shortcut$(n - (pivo - 1))$
14: } `fim se`

Isso é muito interessante pois poupa tempo e esforço mantendo a fidelidade dos testes em relação ao fator de medição de interesse. Esse fator foi, no caso, o número de instruções (número de comparações, no caso de algoritmos de ordenação). O número de instruções é o fator mais convincente a ser medido para quem deseja comparar a experimentação com uma análise assintótica.

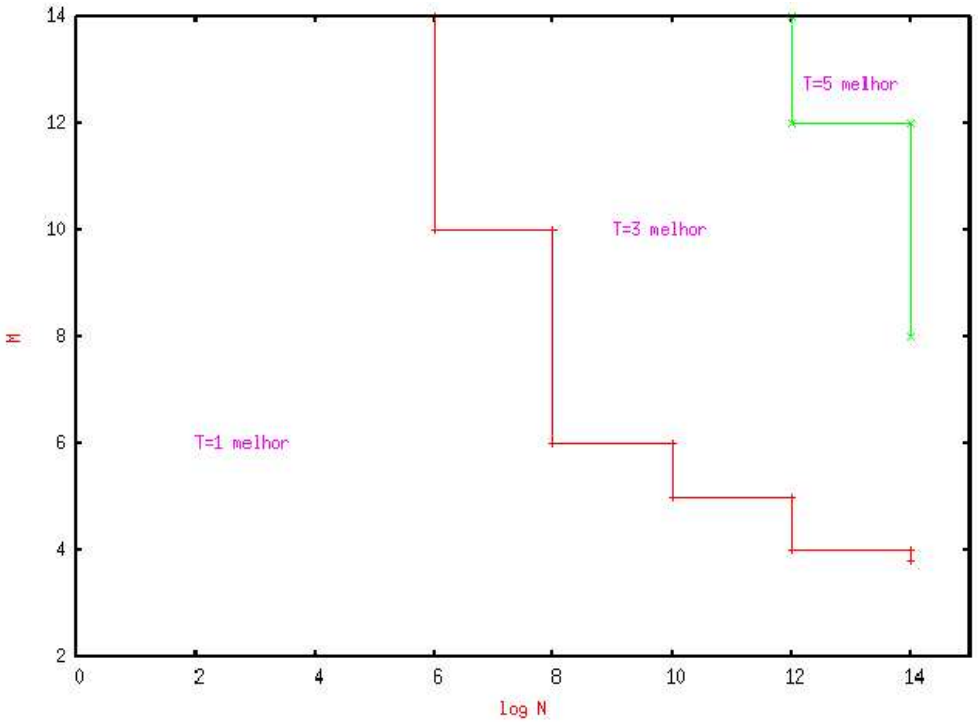

Figura 2.5: Melhor T em função de n e m.

Resultados obtidos para estratégias de T fixo: Podemos observar na Figura 2.5 que se mantivermos um mesmo M (quando parar e chamar o *InsertionSort*), quanto maior for o n, maior é o T (não linearmente). A melhoria (menor número de instruções) entre T igual a um e T igual a três é significativa, enquanto de T igual a três para T igual a cinco já é pouca. McGeoch [McG86b] afirma também que a melhoria para maiores valores de T é muito pequena e não significativa.

Variando o T para minimizar o número de comparações: McGeoch [McG86b] determinou uma função que calcula um T bom para uma subseqüência de tamanho n. As Figuras 2.6, 2.7 e 2.8 mostram os resultados obtidos para encontrar um T bom.

N	T
1	1
35	3
93	5
197	7
337	9
515	11
730	13
984	15
1274	17
1603	19
1968	21
2372	23
2813	25

Figura 2.6: T em função de n.

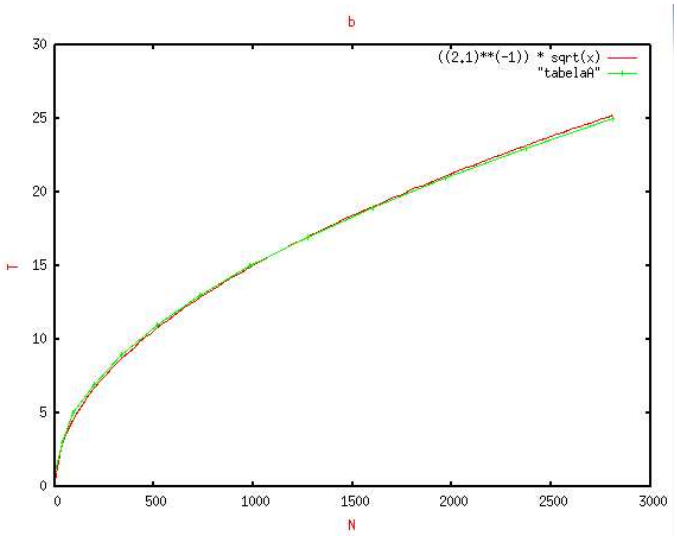

Figura 2.7: T em função de n.

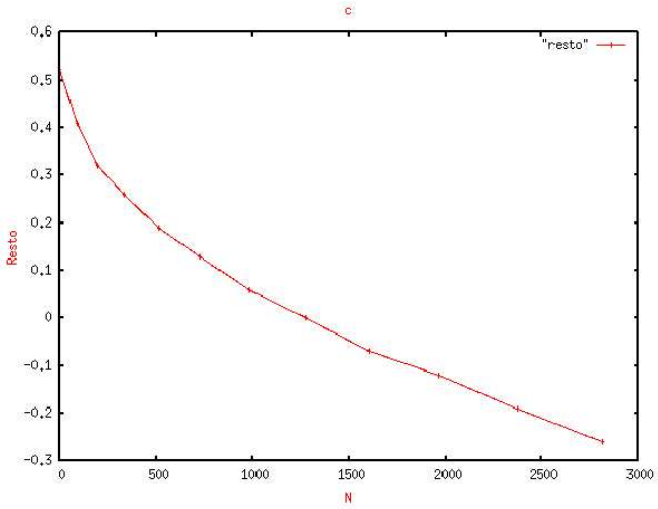

Figura 2.8: Resto.

A figura 2.6 nos dá um valor mínimo de n correspondente para cada $T(n)$, para $n \leq 3.000$. A segunda e a terceira linha da tabela, por exemplo, indicam que 3 é uma escolha ótima de T para subseqüências de tamanho entre 35 e 92. A figura 2.7 mostra a tabela da figura 2.6 de forma gráfica, juntamente com o gráfico da função $(2,1)^{-1}\sqrt{n}$. Já a figura 2.8 mostra a sobra do arredondamento da função $(2,1)^{-1}\sqrt{n}$ para chegarmos nos números da tabela da figura 2.6.

Em [McG86b], não foram apresentados dados sobre os números de instruções ocorridos para T fixo nem para T ajustável (ótimo), apenas foi dito quando o número de instruções seria melhor ou pior.

2.6 Otimizações dependentes da arquitetura

A diferença na velocidade de acesso da memória principal para a velocidade de acesso aos *caches* é muito grande (o acesso ao *cache* é mais de 100 vezes mais rápido), portanto pode ser interessante nos preocuparmos com o *cache* durante a experimentação de algoritmos. Vamos definir alguns conceitos importantes. Falta no *cache* ou *cache miss* é o evento "uma informação **não** foi encontrada no *cache* quando solicitada". Acerto no *cache* ou *cache hit* é o evento "uma informação

foi encontrada no *cache* quando solicitada".

Existem basicamente três formas de organizar o *cache*:

- *Cache-mapeado-diretamente*: É a forma mais simples de *cache*, onde a localização de um dado endereço de memória é determinada pelos *bits* menos significativos, ou seja, uma posição de memória k pode ser alocada no endereço $k \mod t$ do *cache*, onde t é o número de blocos do *cache*. Nesse esquema, não há escolha de qual bloco deve ser liberado quando ocorre uma falta no *cache* sendo que há apenas um lugar no *cache* para qualquer bloco de cada vez. Esse esquema simples tem a desvantagem de que, se o programa requer informação de endereços que são mapeados para o mesmo bloco de *cache*, esse terá que ser substituído ocorrendo então uma falta por conflito, mesmo que o *cache* ainda não esteja cheio.

- *Cache-totalmente-associativo*: É o *cache* onde uma informação de qualquer lugar da memória pode ser alocada em qualquer lugar do *cache*. Essa forma de *cache* é mais custosa (tempo) de ser gerenciada. Porém resolve o problema das faltas por conflito *(conflict misses)*.

- *Cache-associativo ("set-associative-cache")*: É um intermediário entre *cache*-mapeado-diretamente e *cache*-totalmente-associativo onde cada endereço da memória pode ser mapeado em alguns locais do *cache*.

  O espaço do *cache* é dividido em $\frac{m}{k}$ partes, onde m é o tamanho (quantidade de blocos) do *cache* e k é o grau de associatividade do *cache*. Cada parte tem k blocos de tamanho. Um *cache*-mapeado-diretamente pode ser descrito como *"one-way set associative"*. Um *cache*-totalmente-associativo é um *k-way set associative* quando k é o tamanho total do *cache*. Ver exemplo de *4-way set associative cache* (Figura 2.9).

  Estudos de desempenho têm mostrado que essa é geralmente a melhor forma de *cache* e que *caches* de *2-* até *16-way set associative* tem desempenho (pouca ocorrência de falta por conflitos) quase igual a de *cache*-totalmente-associativo e com um pequeno custo (tempo) a mais do que *cache*-mapeado-diretamente [LL97].

Em geral, o *Quicksort* faz um bom aproveitamento do *cache*. Isso se dá devido ao tipo de acesso que o *Quicksort* faz durante o particionamento. Embora existam dois marcadores indexando a seqüência, um do começo para o fim e outro do fim para o começo, o acesso é seqüêncial em ambas

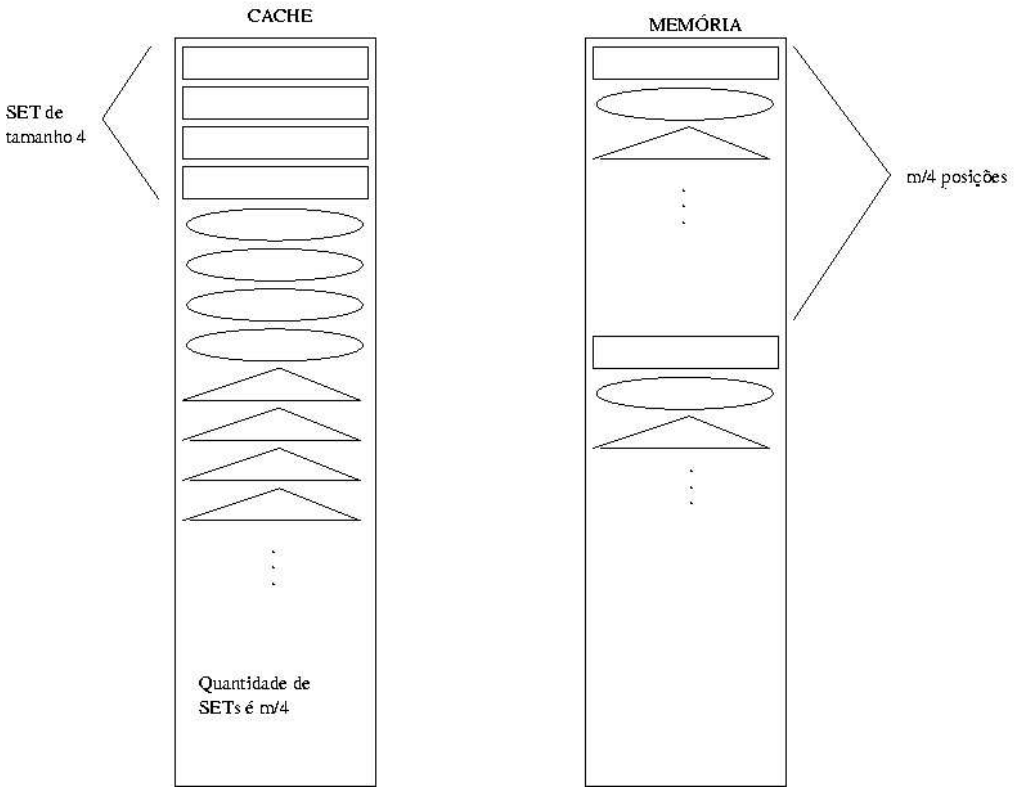

Figura 2.9: 4-way set associative *cache*.

as extremidades da seqüência. Sendo assim, todas as posições de um bloco que é colocado no *cache* são utilizadas, o que caracteriza um excelente aproveitamento espacial do *cache*. Como a cada nível de recursão as posições são utilizadas novamente, também existe um bom aproveitamento temporal do *cache*.

Por exemplo, para um *cache* de tamanho 256KB, cabem 16 números *long int* em um bloco de *cache*. Para entender isso, perceba que o tamanho do bloco do *cache* é de 64 bytes, uma variável *long int* tem 32 *bits* ou 4 bytes, então o tamanho do bloco do *cache* é 16 *long int*. Assim, quando o *Quicksort* usa a primeira posição da seqüência, as 15 seguintes são carregadas no mesmo bloco do *cache*, e essas 15 seguintes muito provavelmente serão lidas (exceção ao encontro dos ponteiros).

Podemos dizer então que se uma subseqüência a ser ordenada for pequena o suficiente para caber inteiramente no *cache* (65536 números de tamanho "long int" cabem no *cache* de 256KB), então haverá no máximo uma falta no *cache* por bloco, que é a falta responsável pelo carregamento do bloco no *cache* caso esse bloco ainda não esteja no *cache*.

LaMarca apresentou em [LL97] duas possíveis otimizações, o *Multi-Quicksort* e o *Memory-Tunned-Quicksort* .

2.7 Resultados experimentais relacionados ao *cache*

Nesta seção comentaremos apenas as otimizações relacionadas ao *Quicksort*, mesmo que os estudos tenham envolvido também outros algoritmos de ordenação. Ressaltamos também que esta seção não é o foco principal deste trabalho, o enfoque principal é continuado no capítulo 3.

2.7.1 Influência do *cache*

Anthony LaMarca e Richard E. Ladner investigaram em [LL97] o efeito que o *cache* pode ter sobre o desempenho de algoritmos de ordenação. Realizaram a análise experimental e também a análise teórica dos algoritmos escolhidos. Estudaram os algoritmos: *MergeSort*, *Quicksort* e *HeapSort*, e foram criadas versões alternativas desses algoritmos com o propósito de otimização voltada ao melhor aproveitamento do *cache* .

Para os três algoritmos testados, as versões otimizadas (para melhor aproveitar o *cache*) tiveram tempo total de execução menor do que as versões originais.

Concentraram-se em medir:

- número de instruções (comparações);
- faltas no *cache* ;
- tempo de execução.

Otimizações relacionadas ao *Quicksort*

Duas otimizações foram feitas para o *Quicksort*, são elas:

- *Multi-QuickSort*: no primeiro nível de recursão do algoritmo a seqüência de entrada deve ser dividida em múltiplas partes para que, estatisticamente, as partes resultantes devam caber no *cache* . Dessa forma, o primeiro passo torna-se um pouco mais lento, porém todos os outros níveis de recursão devem ter seqüências que cabem no *cache* , o que torna o resto do processamento muito mais rápido.

- *Memory-Tunned-QuickSort*: Levando em consideração o ponto de vista de otimizar a utilização do *cache* , LaMarca mostrou que é melhor chamar diretamente o *Insertionsort* quando as pequenas subseqüências forem encontradas pois estas acabaram de passar por um particionamento, e portanto ainda devem estar no *cache* , e desse modo, o *Insertionsort* será executado em menos tempo.

LaMarca e Ladner [LL97] tiveram a intenção de mostrar que as faltas no *cache* podem custar (tempo) muito caro, e esse prejuízo (tempo) vai aumentando conforme os processadores se tornam mais rápidos, pois a diferença na velocidade de acesso do *cache* para a velocidade de acesso da memória vai aumentando. Então, escolher o algoritmo mais rápido para resolver um problema envolve entender o desempenho do *cache* .

Porém não podemos deixar de perceber que otimizar um algoritmo em relação ao *cache* é acabar com a portabilidade desse algoritmo, pois o mesmo só funcionará bem em ambientes exatamente iguais.

Foi utilizada uma seqüência de 4.000 a 4.096.000 posições de inteiros de 64bits escolhidos aleatoriamente (distribuição uniforme de probabilidades). Todos os testes foram executados em um *DEC Alphastation 250* e foi simulado um *cache* com capacidade de 2MB com blocos de 32bytes.

Resultados obtidos

Os valores apresentados nos comentários abaixo são valores aproximados, baseados em interpretação visual dos gráficos apresentados em [LL97].

Com relação às variações do *Quicksort* observou-se que enquanto o valor de n manteve as seqüências com um tamanho que cabia completamente no *cache* , não houve diferença significativa entre os três algoritmos testados (*QuickSort, Multi-QuickSort* e *Memory-Tuned-QuickSort*) em

nenhum dos fatores medidos. As diferenças foram observadas somente para n maior que o *cache* , e para os três fatores medidos conforme comentado abaixo:

- número de comparações (instruções): O *Multi-QuickSort* apresentou um aumento significativo (cerca de 25% a mais em relação aos outros dois algoritmos) no número de comparações realizadas, enquanto os outros dois algoritmos se comportaram de forma muitissimo semelhante para este fator de medição.

- Faltas no *cache* : Os três algoritmos se comportaram muito semelhantemente para seqüências de até 1.000.000 de elementos, atingindo uma taxa de $1,2$ faltas no *cache* por elemento. Para seqüências maiores que 1.000.000, apenas o *Multi-QuickSort* manteve a mesma média, enquanto os outros dois algoritmos aumentaram significativamente a taxa de faltas no *cache* ao longo do aumento de n.

- Tempo de execução: Os três algoritmos se comportaram de maneira muito semelhante para seqüências de até 500.000 elementos. Para seqüências entre 500.000 e 4.000.000 de elementos, o *Multi-QuickSort* foi um pouco mais lento que os outros dois, mas esse tempo se iguala novamente para seqüências de 4.000.000 de elementos deixando a entender que provavelmente para seqüências acima de 4.000.000 de elementos o *Multi-QuickSort* deve ser ainda mais rápido (gastar menos ciclos) que os outros dois.

2.7.2 Melhor aproveitamento de memória

Li Xiao, Xiaodong Zhang e Stefan A. Kubricht basearam seu trabalho [XZK00] em [LL97] porém, tentaram não só melhorar o desempenho do *cache* , mas também levaram em consideração as faltas por conflito no *cache* e as faltas no TLB (*Translation Lookaside Buffer misses*).

O *Translation Lookaside Buffer* (TLB) é uma tabela que contém referências cruzadas entre o endereço real e o virtual das páginas de memória. É um tipo especial de *cache* , que é acessado quando os outros dois níveis (*cache* L1 e L2) já falharam e é preciso buscar o dado na memória principal. As políticas de substituição são as mesmas, porém ele guarda apenas apontadores. Quando uma falta no TLB ocorre o sistema é forçado a buscar a tradução do endereço na tabela de páginas da memória.

O TLB funciona como uma *"hot list"* ou um índice rápido das páginas da memória principal que tem sido acessadas recentemente. A quantidade de memória que pode ser acessada a partir do *Translation Lookaside Buffer* é conceitualmente chamado de TLB *space*, ou seja, é o número de endereços que podem ser armazenados no TLB multiplicado pelo tamanho da página da memória. O tamanho da página é 4096 bytes ou 4KB. Para um processador Pentium III com 64 endereços no TLB, o tamanho é $64 \times 4K = 256KB$.

Não esqueçamos que o *cache* é uma seqüência de conjuntos, cada conjunto tem uma ou mais linhas. Cada linha tem um bloco de dados.

Tipos de faltas de *cache* :

- Falta compulsória *(Cold or Compulsary Miss)*: Ocorre porque o *cache* está, inicialmente, sem aqueles dados.
- Falta por conflito*(Conflict Miss)*: Em muitos casos os blocos da memória principal só podem ser colocados num pequeno número (pode ser apenas um) de blocos do *cache* . Isso varia de acordo com o nível de associatividade do *cache* . Maiores detalhes sobre políticas de utilização do *cache* foram mostradas na seção 2.6. Então mesmo que o *cache* não esteja cheio, faltas por conflito podem acontecer.
- Falha por falta de capacidade *(Capacity Miss)*: Ocorre quando o conjunto de trabalho (*working set*) é maior do que o *cache* .

Baseado nos algoritmos propostos em [LL97], Li Xiao, Xiaodong Zhang e Stefan A. Kubricht propuseram novas versões dos algoritmos *MergeSort* e *Quicksort*. Nos interessará nesse momento somente as novas propostas com relação ao *Quicksort*.

Otimizações relacionadas ao QuickSort

- *Flash-QuickSort*: No algoritmo *FlashSort*, os valores máximo e mínimo são identificados para determinar-se o intervalo i da seqüência. Então a seqüência é dividida em c classes de forma que a primeira classe contenha os i/c primeiros elementos, a segunda classe contenha os i/c próximos elementos e assim por diante. Para cada classe o *FlashSort* é chamado novamente até chegar em classes de tamanho um. O algoritmo *Flash-QuickSort* apresentado em [XZK00]

é uma combinação dos algoritmos *FlashSort* e do *Quicksort*, onde os dois primeiros passos seguem a lógica do *FlashSort*, e o terceiro passo é o *Quicksort* em cada uma das subpartes.

- *Inplaced-Flash-QuickSort*: Para melhorar o desempenho geral, outra otimização para o *cache* foi adicionada para melhorar o aproveitamento temporal no *Flash-QuickSort*. Nesse novo algoritmo, o primeiro e o terceiro passo são os mesmos do *Flash-QuickSort*. Já no segundo passo, uma seqüência adicional é usada como *buffer* para armazenar temporariamente os elementos permutados. No *FlashSort* original uma variável simples é usada para armazenar temporariamente o elemento a ser trocado. Uma linha (bloco) do *cache* normalmente armazena mais de um elemento. Então, se usarmos uma só variável para armazenar a troca, estaremos minimizando a possibilidade de reutilização daquele bloco do *cache* . O uso de uma seqüência melhora tanto o aproveitamento do *cache* quanto o tempo de resposta do algoritmo.

Foi usado para testes em [XZK00] o ambiente de simulação *SimpleScalar* e foram simuladas quatro máquinas: *SGI O2 workstation, Sun Ultra-5 workstation, Pentium II PC, Pentuim III PC*.

Foram medidos:

- faltas no *cache* de nível um (L1) e nível dois (L2) por elemento;
- faltas no TLB por elemento;
- contagem de intruções por elemento;
- tempo de resposta medido em número de ciclos.

Nos interessará os resultados obtidos para Pentium II e Pentium III.

Resultados obtidos

Os valores apresentados nos comentários abaixo são valores aproximados, baseados em interpretação visual dos gráficos apresentados em [XZK00].

Com relação às variações estudadas do *Quicksort*:

- Instruções por elemento: O *FlashSort* e o *Implaced-Flash-QuickSort* mantiveram-se numa faixa de quatrocentas a quinhentas instruções por elemento durante todo o intervalo de n testado.

O *Memory-Tuned-QuickSort* teve um aumento muito grande no número de instruções conforme n aumenta, ultrapassando 1.000 para n maior que 2.000.000.

- Faltas no *cache* de nível um por elemento: O *Flash-QuickSort* e o *Implaced-Flash-QuickSort* se mantiveram abaixo de três faltas por elemento durante todo o intervalo de n testado. Já o *Memory-Tuned-QuickSort* e o *FlashSort* aumentaram bruscamente para seqüências maiores do que 500.000 elementos. Este resultado está de acordo com os resultados de [LL97] para o *Memory-Tuned-QuickSort*; ([XZK00] pág 12).

- Ciclos por Elemento: Para testes realizados com uma entrada distribuída uniformemente, podemos ordenar os algoritmos (do mais rápido para o mais lento) da seguinte maneira: *Memory-Tuned-QuickSort*, *FlashSort*, *Flash-QuickSort* e *Implaced-Flash-QuickSort* ([XZK00] pág 15).

O trabalho realizado por Li Xiao em [XZK00] é interessante, porém os resultados apresentados são um pouco desorganizados, os gráficos apresentam escalas diferentes quando poderiam apresentar a mesma escala para facilitar comparações. Para alguns algoritmos foram apresentados resultados sobre L1 *cache* , para outros foram apresentados resultados para L2 *cache* . Alguns testes variaram n até 8.000.000, outros até 4.000.000 sem justificativas explícitas. A distribuição de probabilidades das seqüências de entrada dos algoritmos também variou de um algoritmo para o outro. Alguns gráficos são redundantes. Toda essa variação dos resultados apresentados torna difícil a comparação entre os algoritmos e com outros possíveis trabalhos.

Capítulo 3

Números pseudo-aleatórios e algoritmos probabilísticos

Eric Bach [Bac91] e Howard J. Karloff [KR93] estudaram a interação entre GNPAs (Geradores de Números Pseudo-Aleatórios) comumente usados e algoritmos probabilísticos familiares como algoritmos de ordenação e seleção. Este estudo é muito interessante pois quando se considera o desempenho do algoritmo em função do número de instruções, o impacto é muito mais significativo e mais convincente do que pequenas diferenças de tempo de resposta causadas pela influência do *cache*.

Karloff assumiu, assim como Bach [Bac91], que uma semente perfeitamente aleatória está disponível e usa essa semente para gerar números pseudo-aleatórios. Esses números gerados são então usados pelos algoritmos probabilísticos.

Considere as restrições abaixo para um algoritmo de *Quicksort*:

(a) Quando tivermos dois problemas a ordenar recursivamente, trabalhamos primeiro no menor.

(b) Cada subproblema de tamanho maior ou igual a 1 invoca o algoritmo de particionamento

usando um número pseudo-aleatório, ou seja, para um problema de tamanho n, n números pseudo-aleatórios serão usados.

(c) Usaremos partições estáveis (significa que os elementos das subseqüências terão sempre a mesma ordem relativa da seqüência anterior a ela, ou seja, se um elemento menor que o pivô estava a esquerda de outro elemento menor que o pivo, esse continuará a sua esquerda).

Um gerador por números aleatórios pode ser dito por congruência linear se for da forma $f(x) = (ax + c) \mod m$ [Knu00a] onde m é o tamanho do período desejado. Para garantir que esse período seja realmente o escolhido m tem que ser um número primo.

O *Quicksort* probabilístico padrão usa n números aleatórios, então podemos dizer que ele usa $n \log n$ bits aleatórios já que cada número precisa de $\log n$ bits para ser representado. O *Quicksort* probabilístico com gerador por congruência linear usa apenas um número aleatório como semente para gerar os outros números pseudo-aleatórios. Podemos então dizer que esse algoritmo usa $\log n$ bits aleatórios.

Considere um gerador por congruência linear seguindo as seguintes restrições:

(i) a semente é escolhida aleatoriamente entre $\{0, 1, ..., m-1\}$;

(ii) $m = m(n) > n$: m é escolhido em função de n e tem que ser maior que n, pois é desejável que o período dos números pseudo-aleatórios esteja ao menos na mesma escala de grandeza que n;

(iii) $(a, m) = 1$: a e m são primos entre si;

(iv) $c = 0$ ou $(c, m) = 1$;

(v) o período do gerador é maior do que $n/4$.

Teorema 7. *Para uma variação do Quicksort que satisfaça as condições (a)—(c), usando um gerador por congruência linear que respeite as restrições (i)—(v), existe pelo menos uma permutação de entrada que requer tempo* $\Omega\left(n^4/m^2\right)$.

Prova: A idéia é construir uma permutação A de $\{0, 1, ..., n-1\}$ para a qual existem $\lceil n^2/(16m) \rceil$ sementes especiais para as quais o tempo requerido é de pelo menos cn^2.

A seqüência A será construída de forma que quando x for uma dessas $\lceil n^2/(16m) \rceil$ sementes, o algoritmo irá "gastar" todos os números pseudo-aleatórios até achar o número 1 como número pseudo-aleatório, porém isso acontecerá após $\lfloor n/4 \rfloor$ iterações. Além disso, após esse "gasto", o algoritmo começa então a particionar a "célula mais a esquerda" da seqüência original, essa subseqüência terá o mesmo problema (partição estável) porém de tamanho $\lfloor n/4 \rfloor$, ou seja, ainda é um problema grande.

Uma vez que provarmos isso faltará então apenas provar que para ordenar uma seqüência de tamanho $\lfloor n/4 \rfloor$ requer tempo $\Omega\left(n^2\right)$ para qualquer uma das $\lceil n^2/(16m) \rceil$ sementes.

Primeiramente assinalamos $A[0] = \lfloor n/4 \rfloor$, ou seja, assinalamos a primeira posição do vetor a ser ordenado com um número muito próximo ou igual a $n/4$.

Ache um x_0, de forma que $f^{\lfloor n/4 \rfloor}(x_0) = 1$.

Pode-se achar esse x_0 usando a função inversa da função do gerador por CL, recursivamente, $\lfloor n/4 \rfloor$ e começando com 1.

Assuma que $f^{(0)}(x)$ é o *Quicksort* em seu primeiro nível de execução (iteração ou recursão) com semente x para o gerador CL, assuma também que

$$f^{(i)}(x) = f(f^{(i-1)}(x)) \text{ e que } y_i = f^{(i)}(x_0),\ 0 \leq i \leq \lfloor n/4 \rfloor$$

então

$$y_{\lfloor n/4 \rfloor} = 1\,.$$

Isso significa que o $(n/4)$-ésimo número aleatório gerado a partir da semente x_0 será igual a 1, e essa será a primeira ocorrência de 1. Considere que $Y = y_0, y_1, y_2, \ldots, y_{\lfloor n/4 \rfloor}$, são os $\lfloor n/4 \rfloor$ primeiros números pseudo-aleatórios gerados quando a semente é x_0. Considere também que L é a quantidade de elementos da subseqüência corrente para uma partição qualquer. Definimos então que $\text{hash}(y, L) = \lfloor L * y/m \rfloor$ e também que $\text{hash}(y) = \text{hash}(y, n)$. Hash é o índice, no vetor, do pivô da vez. Podemos dizer que

$$\text{hash}(y) = \left\lfloor \frac{n}{m} * y \right\rfloor$$

sendo assim, a razão m/n determina a quantidade de valores de y que teremos com o mesmo

hash (mesma posição final em A)

$$|\{y \in 0, 1, ..., m-1 \colon \text{hash}(y) = u\}| \leq \lceil m/n \rceil, \ (\forall u).$$

Vamos dizer que $K = \{\text{hash}(y_0), \text{hash}(y_1), ..., \text{hash}(y_{\lfloor n/4 \rfloor})\}$. Sabemos que m não precisa ser muito grande, basta que ele seja maior que n para que o período seja também maior que n. Podemos assumir que $m < n^2/16$. Temos $n/4$ números no total (no conjunto K), porém considerando que teremos números repetidos numa razão de m/n, temos

$$|K| = \frac{n/4}{m/n} = \frac{n^2}{4m} \geq \frac{n^2}{16m}.$$

Então, existe um conjunto de elementos de tamanho $k = n^2/16m$ que tem seus respectivos hash(y) distintos. Esse conjunto é o conjunto das sementes especiais $\{y_{i_1}, y_{i_2}, ..., y_{i_k}\}$.

Colocamos nessas posições valores grandes (pelo menos $3n/4$) para que, quando a partição ocorra, esses números pseudo-aleatórios serão "gastos" deixando ainda uma seqüência de tamanho $\lfloor n/4 \rfloor$. Por exemplo, sendo s uma semente especial e i o número da iteração corrente:

$$A[\text{hash}(y_{i_j})] = n - (\lfloor n/4 \rfloor) - i_j$$

onde $1 < j < k$.

Existe uma permutação

$$\sigma \text{ de } \{0, 1, ..., \lfloor n/4 \rfloor\} \tag{3.1}$$

que requer tempo cn^2, é a permutação de pior caso. Posicionemos então, teoricamente, esses $\lfloor n/4 \rfloor + 1$ próximos elementos

$$\sigma(0), \sigma(1), ..., \sigma(\lfloor n/4 \rfloor)$$

de forma a gerar $\lfloor n/4 \rfloor$ próximas partições péssimas, ou seja, mais à esquerda possível.

Até agora alocamos $k + 1 + \lfloor n/4 \rfloor$ elementos. Ou seja, são aproximadamente $2 + 5n/16$ elementos alocados. As outras entradas em A podem ser preenchidas arbitrariamente.

Se rodarmos o *Quicksort* em A, o primeiro pivô escolhido será um número grande ($A[\text{hash}(s)] = n - (\lfloor n/4 \rfloor) - i \geq 3n/4$). Isso fará com que os próximos $\lfloor n/4 \rfloor$ números aleatórios sejam "gastos"

na parte direita (menor) da primeira partição.

Após isso, o algoritmo começará então a resolver o lado esquerdo (maior) da seqüência original; mas agora o número pseudo-aleatório gerado é 1, e a seqüência gerada a partir desse 1 é σ (3.1). Tendo usado partições estáveis, nós garantimos que nesse momento os elementos que posicionamos anteriormente em locais "chave" continuam nas mesmas posições. Sendo assim, as próximas partições ocorrerão usando os termos de σ e gastando assim tempo quadrático.

Então, para cada uma das $n^2/(16m)$ sementes, o tempo requerido é pelo menos cn^2. Assim, o tempo médio em m sementes é de

$$n^2/(16m) \cdot cn^2 \cdot 1/m = \frac{cn^4}{m^2}.$$

Para $m > n^2/16$ o tempo de resposta seria $O(1)$, mas sabemos que o tempo de resposta no *Quicksort* não pode ser melhor do que $\Theta(n \log n)$. Resolvendo a inequação $\frac{n^4}{m^2} \leq n \log n$ concluiremos que m na verdade não pode ser (ou não fará diferença se for) maior do que $n^2/(\log n)^2$. Se tivermos $m = O(n)$ o tempo de resposta (instruções) pode ser quadrático.

□

Na prova anterior, existem $\lceil (11n/16) - 2 \rceil !$ permutações possíveis que completam A, sendo assim, existem $\lceil (11n/16) - 2 \rceil !$ permutações que têm tempo $\Omega\left(\frac{n^4}{m^2}\right)$.

***Gerador X-independente*:** Um gerador por números aleatórios é considerado X-independente quando X é o número de sementes realmente aleatórias usadas pelo algoritmo. Esse número influencia na aleatoriedade da seqüência pseudo-aleatória gerada. Embora pareça redundante, é importante reforçarmos que uma seqüência gerada por um gerador X-independente é chamada de seqüência X-independente. Por exemplo, se escolhermos aleatoriamente A e B, temos então uma pequena seqüência (de dois elementos) que é bi-independente. Caso geremos C, onde $C = A \cdot B$, temos agora uma seqüência com três elementos, porém continua sendo bi-independente. Podemos dizer então que $X_0, X_1, ..., X_n$ são variáveis aleatórias t-independentes se cada subconjunto de t variáveis forem mutuamente independentes.

Karloff então sugere uma versão probabilística do *Quicksort* que também precisa de apenas $O(\log n)$ bits aleatórios, e que usa um gerador não linear e penta-independente. O objetivo dessa

versão foi fazer um *Quicksort* aleatório onde não exista uma relação entre o gerador por números aleatórios e o algoritmo que possa gerar tempo quadrático.

Lema 1. *Consideremos que n é primo,* $2 \leq t \leq n$, $\{a_0, a_1, ..., a_{t-1}\} \in \{0, ..., n-1\}$ *são escolhidos aleatória e independentemente, e que* $X_i = a_0 + a_1^i + a_2^{i^2} + ... + a_{t-1}^{i^{t-1}}$. *Sendo assim,* X_i *tem distribuição uniforme em* $\{0, ..., n-1\}$ *e os elementos* $X_0, ..., X_{n-1}$ *são t-independentes. Dado* $X_i = j$, $\{X_j | j \neq i\}$ *é (t-1)-independente e* X_j *tem distribuição uniforme em* $\{0, ..., n-1\} (\forall j \neq i)$.

3.1 *QS-5i*

O *QS-5i* é um tipo de *Quicksort* aleatório e iterativo proposto por Karloff para n primo [KR93]. Funciona escolhendo aleatoriamente cinco números a, b, c, d, e pertencentes ao conjunto $\{0, 1, ..., n-1\}$. Depois disso, usa esses cinco números para escolher o pivô ($pivo \leftarrow (a + bi + ci^2 + di^3 + ei^4)$ mod n) e então `particiona` utilizando esse pivô. Repete esses passos n vezes. A intenção é que ao final dessas n escolhas pseudo-aleatórias do pivô, a maioria dos elementos da seqüência já tenham sido escolhidos como pivô. Estamos assumindo que a função `particiona` já existe e que essa não faz nada se receber pivôs repetidos. Depois de tudo isso, o algoritmo chama a função *clean-up*. O *clean-up* consiste em particionar usando os pivôs que não foram escolhidos durante todas as iterações de escolhas aleatórias, pois $(X_0, X_1, ..., X_{n-1})$ (escolhas feitas no laço de iterações) normalmente não é uma permutação de $(0, 1, 2, ..., n-1)$.

Podemos ter a impressão de que esse algoritmo será sempre quadrático, expliquemos então o porquê dessa impressão estar errada. Vamos chamar de A o evento *a função* `particiona` *é chamada para subseqüências de tamanho um* e esse evento tem custo (número de comparações) zero. Assim como em qualquer versão do *Quicksort*, quanto mais equilibrado forem as partições maior o número de eventos A ocorrendo na árvore e, consequentemente, mais cedo terminará a ordenação. Por exemplo se o melhor caso ocorrer, após $\log_2 n$ iterações todos os pivôs escolhidos vão ocasionar A, ou seja, terão custo zero. Então apesar de termos obrigatoriamente mais do que n chamadas da função particiona, muitas dessas chamadas podem ter custo zero.

Veja o *Quicksort probabilístico com GNPA penta-independente* descrito no Algoritmo 5.

Quicksort probabilístico com GNPA penta-independente (QS-5i) ordena corretamente partindo do princípio de que ele `particiona` a seqüência de entrada em cada um de seus elementos.

Algoritmo 5 QS probabilístico com GNPA penta-independente(seqüência S)

1: Escolha aleatoriamente cinco elementos a, b, c, d, e pertencentes ao conjunto $\{0, 1, ..., n-1\}$
2: $i \leftarrow 0$
3: `enquanto` $i \leq n-1$
4: $\quad pivo \leftarrow (a + bi + ci^2 + di^3 + ei^4) \mod n$ {pivo aleatório}
5: $\quad$ particiona($S[pivo]$) {o `particiona` não faz nada caso receba um pivo que ja recebeu antes}
6: } `fim enquanto`
7: clean-up() {`particiona` em cada índice que não foi particionado anteriormente}

Este algoritmo roda em tempo esperado O($n \log n$) (Teorema 8 abaixo).

Considere que C_i é o custo total da i-ésima iteração do *QS-5i* e que Xj é o pivô pseudo-aleatório gerado na iteração j. Considere também $i \in \{1, 2, ..., n-1\}$ e y pertencente ao conjunto das sementes verdadeiramente aleatórias (*escolhidas uniformemente dentro do conjunto* $\{0, 1, ..., n\}$). Assim queremos mostrar (lema 4) que C_i é O(n/i). Defina

$$R_{i,y} = \min\left(A[Xj] - y \mid 0 \leq j < i \text{ e } y < A[Xj] \leq n\right), e$$

$$L_{i,y} = \min\left(y - A[Xj] \mid 0 \leq j < i \text{ e } A[Xj] < y\right).$$

$R_{i,y}$ é a distância de y até o primeiro pivô mais próximo à direita (dentre todos os pivôs anteriores gerados), ou seja, é a distância entre y e o final desse nodo (subseqüência) que está sendo ordenado na iteração i. $L_{i,y}$ é a distância correspondente para o lado esquerdo.

Note que se $A[X_i] = z$, então $C_i = R_{i,z} + L_{i,z} \ \ (\forall z \in Z_n)$, onde Z_n é o conjunto de todos os elementos da subseqüência corrente da recursão ou iteração.

Lema 2. *Seja k um inteiro positivo qualquer e U uma variável aleatória discreta de forma que tanto $\mu =$ E(U) quanto $E((U-\mu)^k)$ existam. Defina $\tau_k > 0$ como a raiz k-ésima de $E((U-\mu)^k)$. Então, para qualquer $t > 0$,*

$$Pr(|U - \mu| \geq t\tau_k) \leq \frac{1}{t^k}.$$

Denotamos por $E((U-\mu)^k)$ o k-ésimo momento central de U. Para o caso especial de $k = 2$, esse momento é a variância de U, e τ_2 o desvio padrão.

Prova: Da igualdade

$$Pr(|U - \mu| \geq t\tau_k) = Pr((U-\mu)^k \geq t^k \tau_k^k)$$

é suficiente provar

$$Pr\left((U-\mu)^k \geq t^k \mathrm{E}\left((U-\mu)^k\right)\right) \leq \frac{1}{t^k}\,.$$

Considere $x' = (U-\mu)^k$. Reescrevendo a equação acima temos

$$Pr\left(x' \geq t^k \mathrm{E}(x')\right) \leq \frac{1}{t^k}$$

que segue da desiqualdade de Markov. □

Lema 3. *Para qualquer $i \in \{1, 2, ..., n-1\}$, qualquer inteiro positivo r, e qualquer $y, z \in Z_n$,*

$$Pr\left(R_{i,y} > r \mid A[X_i] = z\right) \leq \frac{4n^2}{i^2 r^2}\,,\ Pr\left(L_{i,y} > r \mid A[X_i] = z\right) \leq \frac{4n^2}{i^2 r^2}\,.$$

Prova: Vamos assumir $r < n - y$ (dentro do intervalo da partição atual). Pela definição de $R_{i,y}$, podemos afirmar que $Pr(R_{i,y} > n - y) = 0$ é sempre verdadeiro. Vamos considerar também que $S = \{y+1, y+2, ..., y+r\}$. Note que $S \subseteq Z_n$, uma vez que $r < n - y$. Então

$$Pr(R_{i,y} > r \mid A[X_i] = z) = Pr\left(\bigwedge_{j=0}^{i-1}(A[X_j] \notin S) \;\middle|\; A[X_i] = z\right), \text{ e}$$

$$Pr\left(\bigwedge_{j=0}^{i-1}(A[X_j] \notin S) \mid A[X_i] = z\right) \leq \frac{4n^2}{i^2 r^2}\,. \tag{3.2}$$

Perceba que a afirmação acima é sempre verdadeira se $ir \leq n$, pois o lado direito da inequação será incondicionalmente maior do que 4. Conseqüentemente , assuma $ir > n$. Repare que $r = |S|$. Para $j \in \{0, 1, ..., i-1\}$, considere que

$$Y_j = \begin{cases} 1 & \text{se } A[X_j] \in S \\ 0 & \text{caso contrário,} \end{cases}$$

e que

$$p = \mathrm{E}(Y_j \mid A[X_i] = z) = Pr(A[X_j] \in S \mid A[X_i] = z) = r/n\,,$$

uma vez que X_j é uniformemente distribuído e independente de X_i. Considere $U = \sum_{j=0}^{i-1} Y_j$.

Então, o lado esquerdo da inequação 3.2 pode ser escrito como $Pr(U = 0 \mid A[X_i] = z)$.

$$\mu = \mathrm{E}(U \mid A[X_i] = z) \;=\; \mathrm{E}\left(\sum_{j=0}^{i-1} Y_j \mid A[X_i] = z\right) \;=\; \sum_{j=0}^{i-1} \mathrm{E}(Y_j \mid A[X_i] = z) = ip\,.$$

Defina $\tau = \mathrm{E}((U-\mu)^4 \mid A[X_i] = z)$

$$Pr(U = 0 \mid A[X_i] = z) \leq Pr(|U-\mu| \geq \mu \mid A[X_i] = z) = Pr(|U-\mu| \geq \frac{\mu}{\tau} \cdot \tau \mid A[X_i] = z)$$

(Lema 2)

$$\begin{aligned}
&=\leq \frac{\tau^4}{\mu^4} = \frac{1}{\mu^4}\mathrm{E}\left((U-\mu)^4 \mid A[X_i] = z\right) = \frac{1}{\mu^4}\mathrm{E}\left(\left(\sum_{j=0}^{i-1}(Y_j - p)\right)^4 \mid A[X_i] = z\right) \\
&= \frac{1}{\mu^4}\left[\sum_{j=0}^{i-1}\mathrm{E}\left((Y_j-p)^4 \mid A[X_i] = z\right) + \binom{4}{2}\sum_{0\leq j<k<i}\mathrm{E}\left((Y_i-p)^2(Y_k-p)^2 \mid A[X_i] = z\right)\right] \\
&= \frac{1}{i^4p^4}\left[i\left(p(1-p)^4 + (1-p)(-p)^4\right) + 6\binom{i}{2}\left(p(1-p)^2 + (1-p)(-p)^2\right)^2\right] \\
&= \frac{ip(1-p)((1-p)^3+p^3) + 6\binom{i}{2}p^2(1-p)^2}{i^4p^4} \leq \frac{ip + 3i^2p^2}{i^4p^4} < \frac{4i^2p^2}{i^4p^4} = \frac{4}{i^2p^2} = \frac{4n^2}{i^2r^2}\,.
\end{aligned}$$

□

Uma vez mostrado (pelo Lema 3) que as afirmações

$$Pr\left(R_{i,y} > r \mid A[X_i] = z\right) \leq \frac{4n^2}{i^2r^2} \text{ e}$$

$$Pr\left(L_{i,y} > r \mid A[X_i] = z\right) \leq \frac{4n^2}{i^2r^2}$$

são verdadeiras, provaremos agora o seguinte:

Lema 4. *O custo esperado de cada uma das duas subpartições geradas em um particionamento i é* $\mathrm{O}(n/i)$.

$$\mathrm{E}(R_{i,y} \mid A[X_i] = z) = \mathrm{O}(n/i), \text{e}\, \mathrm{E}(L_{i,y} \mid A[X_i] = z) = \mathrm{O}(n/i)\,.$$

Prova: O tamanho esperado do lado direito da subseqüência particionada em y na iteração i tal que $A[X_i] = z$ é igual à somatória de n probabilidades de $R_{i,y}$ ser igual a um número de 0 a n

multiplicado por esse mesmo número, tal que $A[X_i] = z$.

$$\mathrm{E}(R_{i,y} \mid A[X_i] = z) = \sum_{s=0}^{n} s \cdot Pr(R_{i,y} = s \mid A[X_i] = z)\,.$$

Trocando a multiplicação por outro somatório, podemos reescrever a equação da seguinte maneira

$$\begin{aligned}
\mathrm{E}(R_{i,y} \mid A[X_i] = z) &= \sum_{s=0}^{n} \sum_{r=0}^{s-1} Pr(R_{i,y} = s \mid A[X_i] = z) \\
&= \sum_{r=0}^{n} Pr(R_{i,y} = s \mid A[X_i] = z) \\
&\leq 1 + \lceil n/i \rceil + \sum_{r=\lceil n/i \rceil + 1}^{n} \frac{4n^2}{i^2 r^2}\,.
\end{aligned}$$

Limitando a somatória acima com uma integral, temos

$$\begin{aligned}
\mathrm{E}(R_{i,y} \mid A[X_i] = z) &\leq 1 + \lceil n/i \rceil + \int_{\lceil n/i \rceil}^{n} \frac{4n^2}{i^2 r^2} dr \\
&\leq 1 + \lceil n/i \rceil + 4n/i - 4n/i^2 \\
&= \mathrm{O}(n/i).
\end{aligned}$$

□

Teorema 8. *Sabendo que o tempo esperado de execução da i-ésima iteração da linha 5 do algoritmo (particiona) é* $\mathrm{O}(n/i)$ *(Lema 4), para* $i \geq 1$*, então o custo total esperado para todas as execuções do* `particiona` *é* $\mathrm{O}(n \log n)$.

Demonstração. Podemos considerar que o custo da primeira partição é $n + 1$, e como o custo das outras partições é $\mathrm{O}(n/i)$, escrevemos a equação do custo total da seguinte maneira

$$C(n) = (n+1) + \sum_{i=1}^{n-1} \mathrm{O}\left(\frac{n}{i}\right) = \mathrm{O}(n + n \sum_{i=1}^{n-1} \frac{1}{i}) = \mathrm{O}(n \log n)\,,$$

pois a séria harmônica em n é limitada em $\ln(n) + \mathrm{O}(1)$ [KGP94]. □

Teorema 9. *O custo total da linha 7 do algoritmo clean-up é* $\mathrm{O}(n)$.

Demonstração. Considere que $D_y = L_{n-1,y} + R_{n-1,y}$. O custo total do *clean-up* é no máximo $\sum_{y=0}^{n-1} D_y$.

$$
\begin{aligned}
E\left(\sum_{y=0}^{n-1} D_y\right) &= \sum_{y=0}^{n-1} E(D_y) = \sum_{y=0}^{n-1} E(L_{n-1,y} + R_{n-1,y}) \\
&= \sum_{y=0}^{n-1}\sum_{z=0}^{n-1} \left\{ \left(E(L_{n-1,y} | A[X_{n-1}] = z) \cdot \left(Pr(A[X_{n-1}]) = z \right) \right) + \right. \\
&\qquad\qquad \left. \left(E(R_{n-1,y} | A[X_{n-1}] = z) \cdot \left(Pr(A[X_{n-1})] = z \right) \right) \right\} \\
&= \sum_{y=0}^{n-1}\sum_{z=0}^{n-1} \mathrm{O}\left(\frac{n}{n-1}\right) \cdot Pr(A[X_{n-1})] = z) \\
&= \sum_{y=0}^{n-1} \mathrm{O}(1) \sum_{z=0}^{n-1} Pr(A[X_{n-1})] = z) = \sum_{y=0}^{n-1} \mathrm{O}(1) = \mathrm{O}(n)\,.
\end{aligned}
$$

□

Capítulo 4

Análise Experimental

Experimentamos cinco versões diferentes do *Quicksort*. Quatro destas foram implementadas neste trabalho. Testamos também a versão implementada pela GNU para fins de comparação.

A seguir listaremos as versões testadas. Os apelidos apresentados em negrito são correspondentes às legendas dos gráficos que teremos ao longo do trabalho:

- *Quicksort* com mediana de 5 **(QS-med5)**,
- *Quicksort* versão da GNU com mediana de 3 **(QS-GNU)**,
- *Quicksort* 5i ou Penta-Independente **(QS-5i)**,
- *Quicksort* com congruência linear **(QS-CL)**,
- *Quicksort* padrão com pivô fixo na primeira posição **(QS-Fixo)**.

As cinco versões testadas são versões iterativas do *Quicksort*. As implementações feitas aqui foram baseadas na implementação da GNU que é uma versão iterativa e usa mediana de 3 para achar o pivô.

Todas as versões testadas têm as seguintes características em comum:

- são iterativas;
- usam o artifício de Sedgewick de remover iterações em pequenas subseqüências (M=4) e chamar o *Insertionsort*;
- usam particionamento duplo (padrão).

Escolhemos quatro critérios de medição para analisar o desempenho dos algoritmos testados:

- tempo de resposta (milisegundos);
- número de instruções;
- números de nodos na árvore de recursão;
- quantidade de trocas realizadas.

Os testes foram realizados em um PC com processador AMD Athlon 1.4GHz, placa mãe K7VTA2 com barramento de 266MHz, 256MB de Memória RAM DDR, Disco Rígido de 7200 RPM, e *cache* de nível 1 (*in chip*) *16-way set-associative* de 256KB com blocos de 64bytes. Todos os testes foram rodados sobre o Sistema Operacional Conectiva Linux 10 com Kernel versão 2.6.5-63077cl. Todos os programas foram desenvolvidos na linguagem C.

Para garantir maior confiabilidade dos resultados, vinte testes foram realizados para cada ponto de cada gráfico e a média destes vinte testes foi utilizada como valor para este determinado ponto. Esse tipo de cuidado é importante pois os resultados dos testes variam de acordo com a disponibilidade dos recursos do sistema operacional e percebemos que vinte testes é um bom número de testes para que a variância seja relativamente pequena entre os resultados. Cada linha do gráfico tem seis pontos de referência onde cada um destes pontos é referente à um valor de n. Os valores de n utilizados foram os seguintes:

- 262.144;
- 2.621.440;
- 5.242.880;
- 7.864.320;

- 10.485.760;
- 13.107.200.

Se uma entrada couber inteiramente no *cache*, a ordenação desta se dará de forma mais rápida. A fim de garantir a comparabilidade dentre os testes, todos os valores de n usados são maiores do que o *cache*. Usamos, como fonte para as seqüências de entrada, arquivos providos pelo *site* `www.random.org`. Nesse site podemos encontrar arquivos com enormes seqüências de bits considerados aleatórios. Utilizando os recursos do site geramos arquivos binários com 1, 10, 20, 30, 40 e 50 *MegaBytes* de bits aleatórios. Com o intuito de permitir a representatividade correta da quantidade de números que obtivemos, cada um desses arquivos grandes foi separado de 32 em 32 bits (4bytes) gerando assim uma seqüência de números *unsigned long int* ($1MB = \frac{1048576}{4} = 262144$ números unsigned long int).

Após realizarmos todos os testes com as entradas aleatórias, percebemos que seria interessante testar também entradas não tão aleatórias, ou seja, organizamos um pouco as seqüências de entrada para observar como isso alteraria o comportamento dos algoritmos. Foi desenvolvido então um novo programa para organizar as seqüências usando mediana de sete. A mediana de sete é um valor que custa caro (tempo) para ser calculado, porém o objetivo desse programa auxiliar não era o de ser rápido, o objetivo era organizar melhor as seqüências de entrada. Esse programa gerou novos arquivos de entrada organizados em aproximadamente 25%, 50% e 75%.

Chamamos de entrada X% organizada uma entrada aleatória que passou por $n \cdot \frac{X}{100}$ iterações com mediana de sete. Tivemos então vinte testes por ponto, seis pontos por linha do gráfico, cinco linhas (algoritmos) por gráfico, e temos dez gráficos. Foram realizados pelo menos $20 \times 6 \times 5 \times 10 = 6000$ testes .

4.1 Trocas

Os próximos quatro gráficos mostram o número de trocas contadas durante a execução dos cinco tipos de *Quicksort* testados variando n dentre os seis valores apresentados na seção anterior.

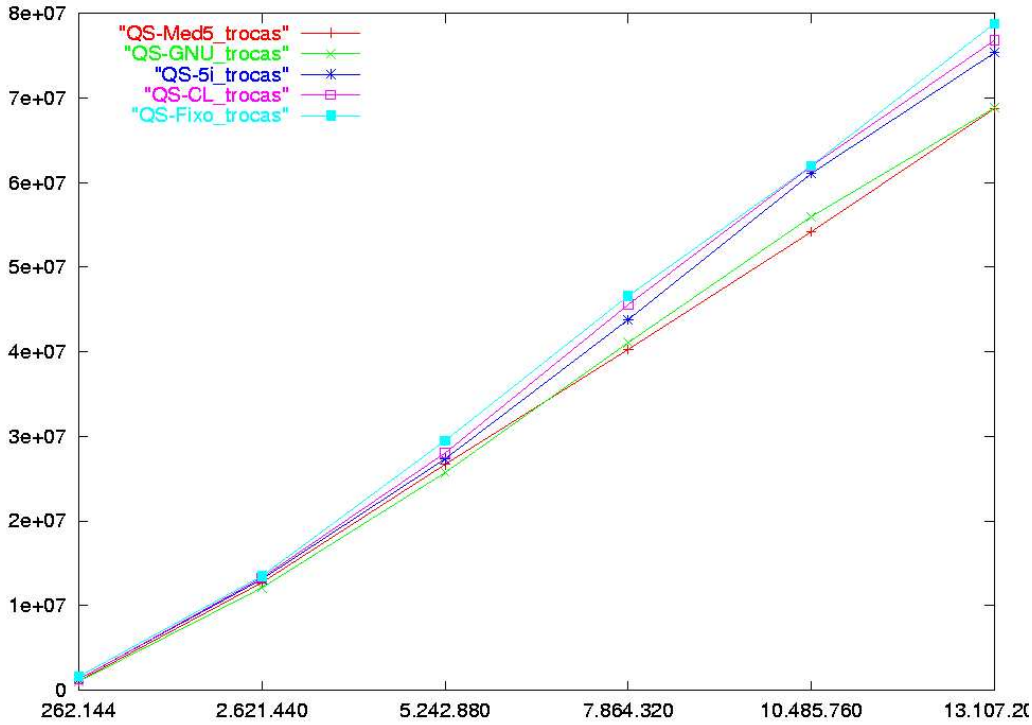

Figura 4.1: Número de trocas, entrada aleatória.

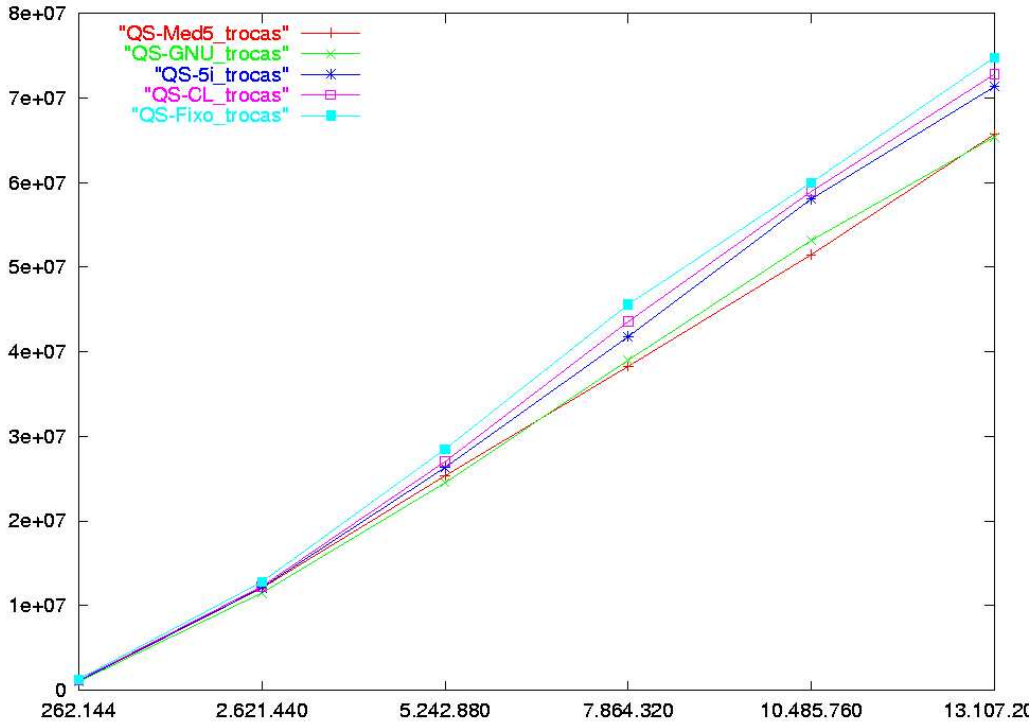

Figura 4.2: Número de trocas, entrada 25% arrumada.

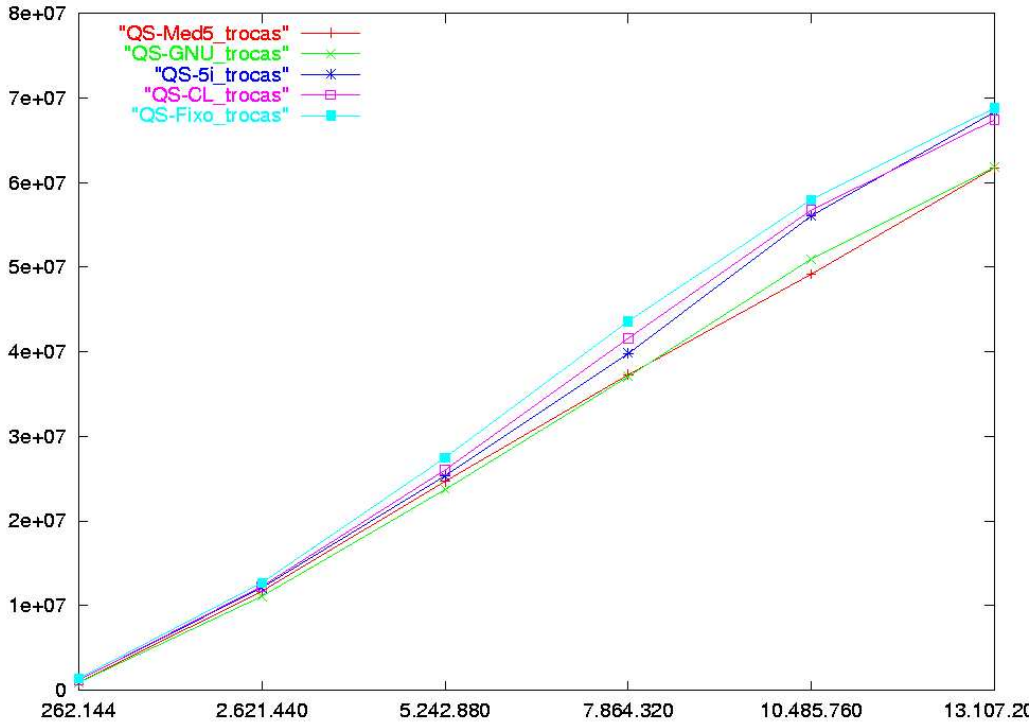

Figura 4.3: Número de trocas, entrada 50% arrumada.

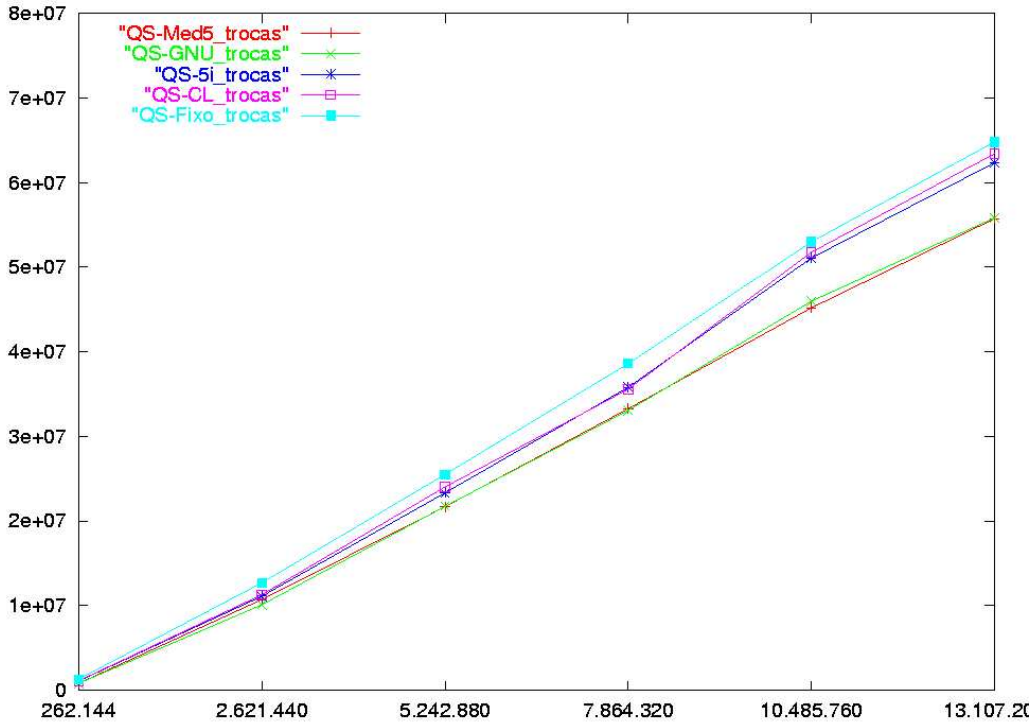

Figura 4.4: Número de trocas, entrada 75% arrumada.

Podemos observar que, na medida em que as seqüências de entrada tiveram maior porcentagem de pré-ordenação, a quantidade de trocas baixou para todos os algoritmos. Porém essa redução na quantidade de trocas não se deu na mesma proporção para todos os algoritmos; para os algoritmos que usam escolha de pivô aleatória a redução foi um pouco maior. Esse fato se da porque se a entrada não é tão aleatória, a aleatoriedade da escolha do pivô tem maior efeito. O intuito da escolha do pivô aleatório fica um pouco prejudicado quando a entrada é também aleatória.

4.2 Número de instruções

A figura abaixo mostra a variação do número de instruções para os algoritmos testados conforme se variou o n.

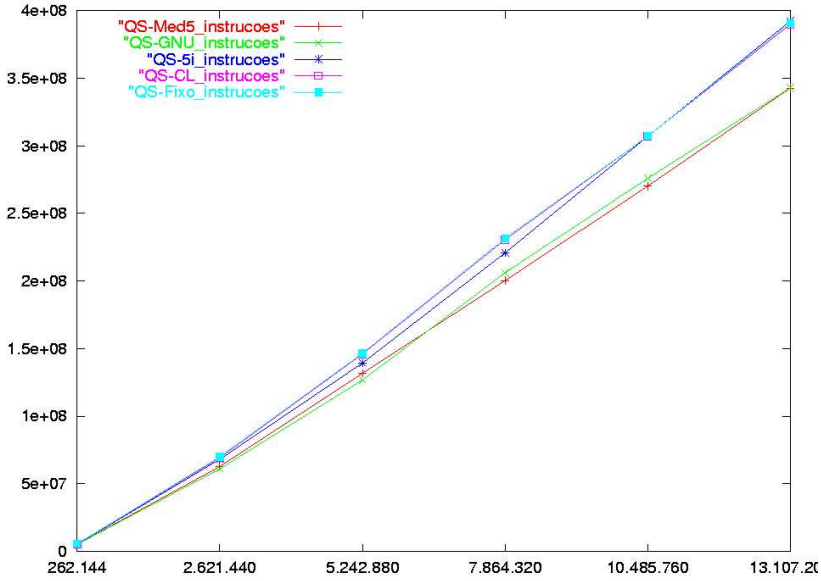

Figura 4.5: Número total de instruções, entrada aleatória.

4.3 Tempo de resposta

Nesta seção apresentamos quatro gráficos referentes aos tempos de respostas medidos em microsegundos para cada um dos algoritmos testados. Cada gráfico contém dados de todos os algoritmos para um mesmo tipo (aleatória ou X% arrumada) de seqüência de entrada.

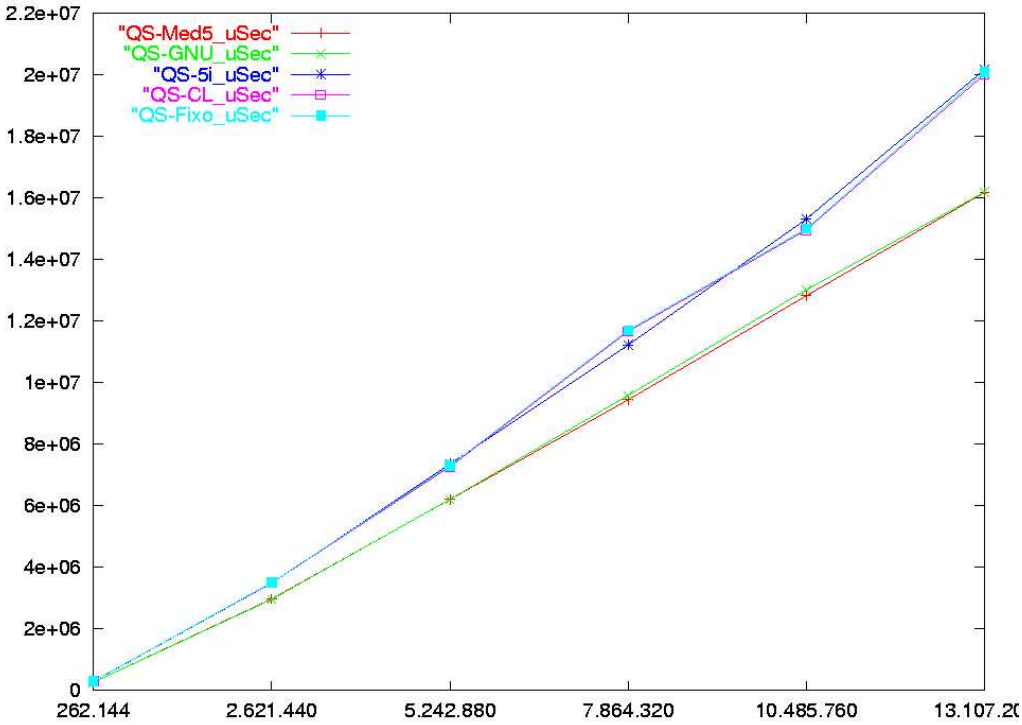

Figura 4.6: Tempo de resposta, entrada aleatória.

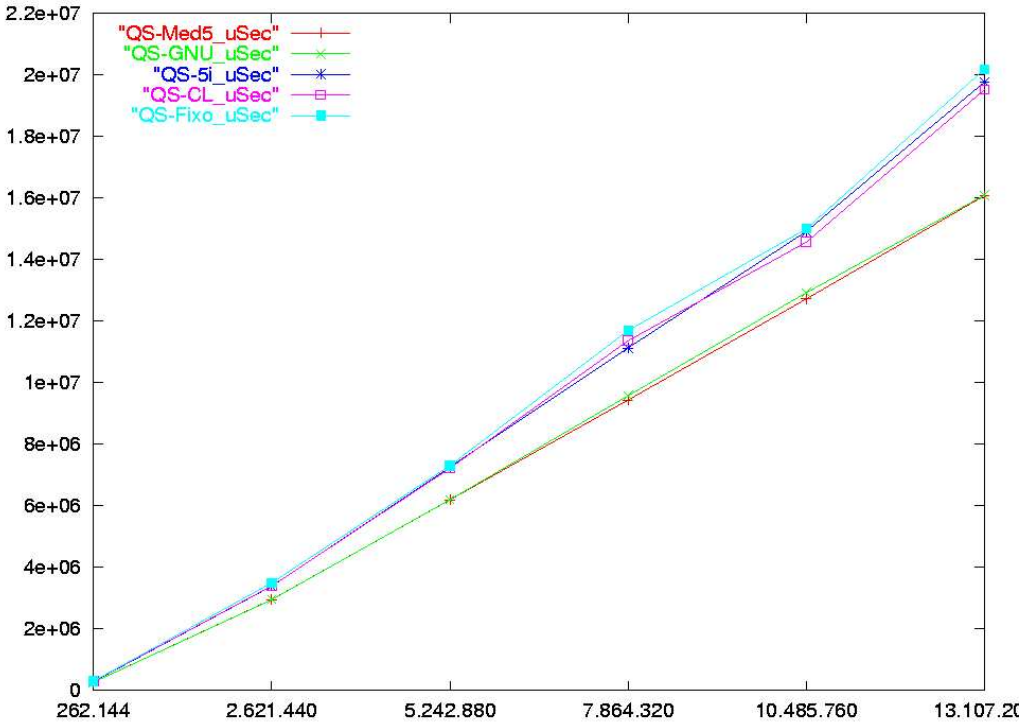

Figura 4.7: Tempo de resposta, entrada 25% arrumada.

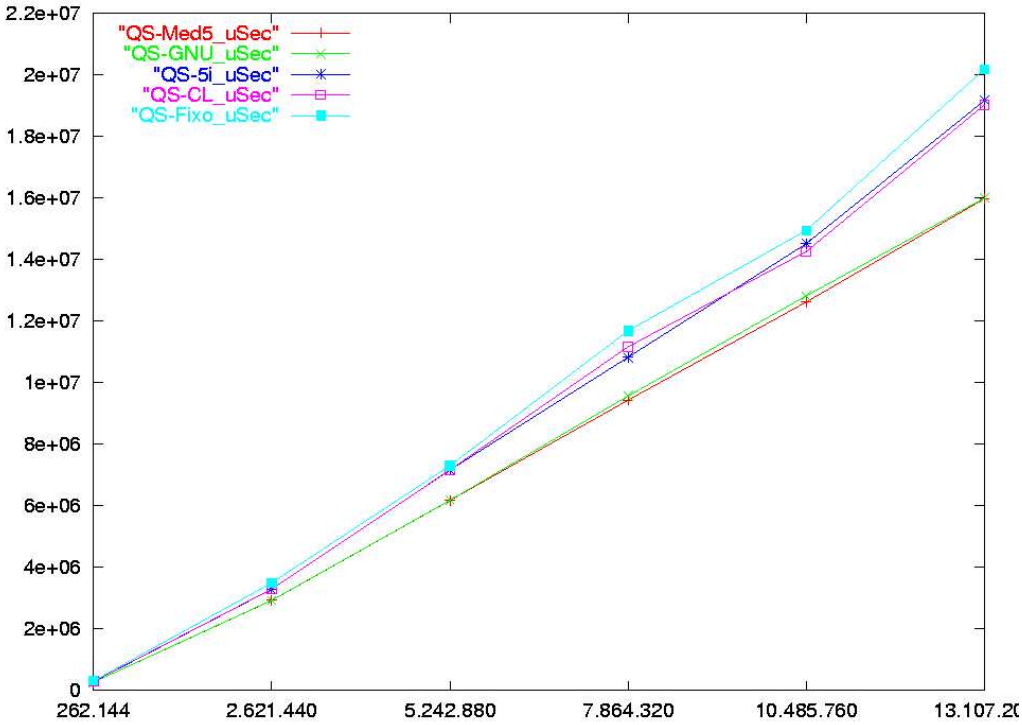

Figura 4.8: Tempo de resposta, entrada 50% arrumada.

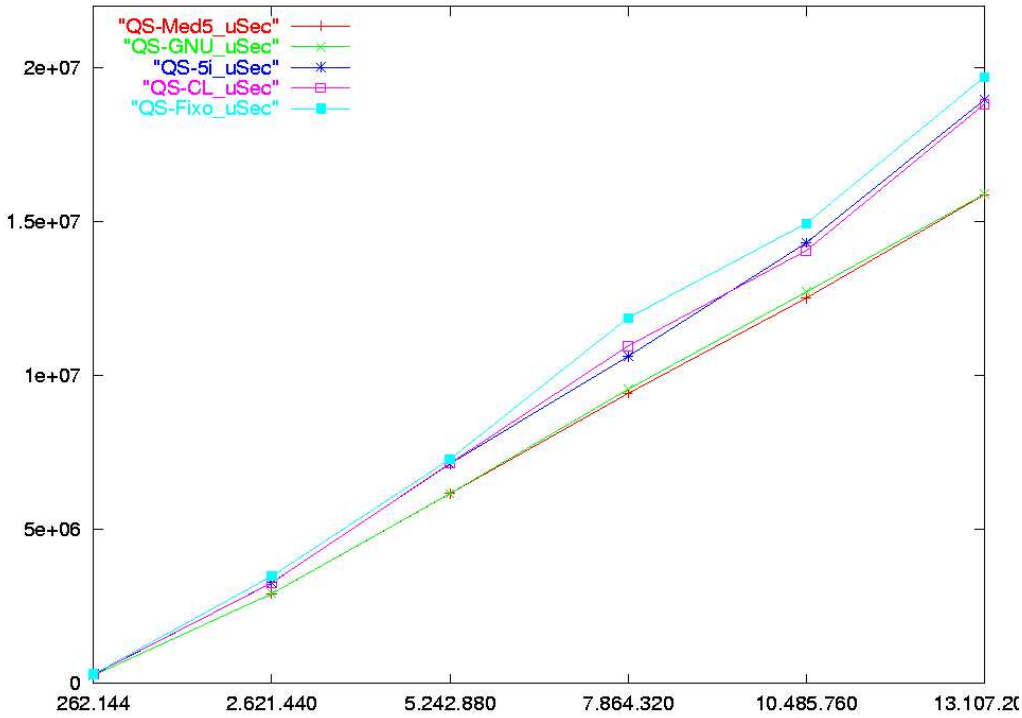

Figura 4.9: Tempo de resposta, entrada 75% arrumada.

O tempo de resposta também diminuiu para todos os algoritmos conforme a seqüência foi sendo arrumada. Porém essa mudança não é tão nítida como nas trocas. Isso acontece porque a quantidade de trocas é responsável apenas por uma pequena porcentagem do tempo de resposta.

4.4 Números de nodos na árvore

A figura abaixo mostra a variação do número de nodos na árvore para os algoritmos testados conforme se variou o n.

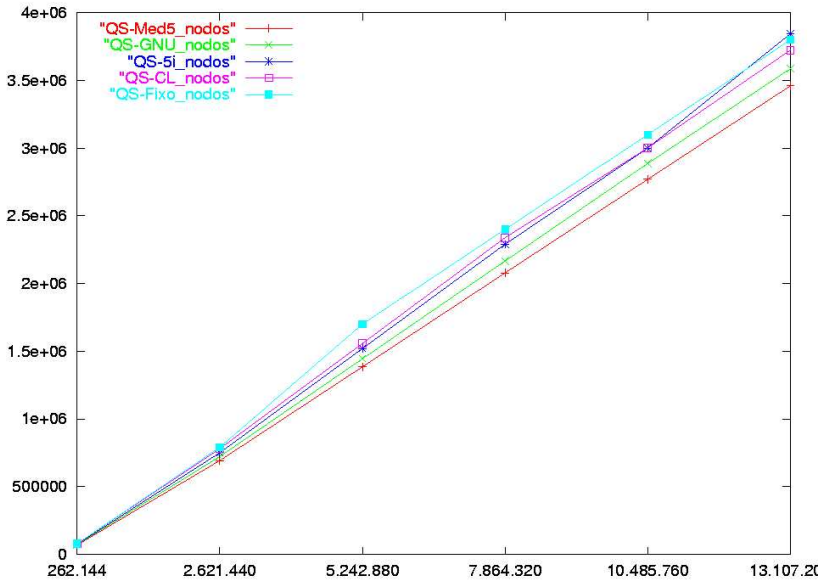

Figura 4.10: Número total de nodos na árvore, entrada aleatória.

O número de instruções e o número de nodos da árvore permaneceram práticamente os mesmos durante os testes com as seqüências arrumadas. A diferença foi tão pequena que seria imperceptível se representada em gráficos.

Capítulo 5

Conclusão

5.1 Comentários Finais

Estudamos neste trabalho diferentes formas nas quais a idéia inicial do *Quicksort* de Hoare pode ser moldada. Dentre elas, o *Quicksort* de Sedgewick, que tem como idéia principal remover as recursões ou iterações desnecessárias no final da execução do algoritmo; o particionamento mediana-de-três, o particionamento mediana-de-T e o particionamento triplo (que só vale a pena ser usado para casos específicos com muitas repetições na entrada).

A iniciativa de otimização pode ser tomada por diferentes pontos de vista como *cache*, instruções, trocas. Estudamos vários artigos correlatos e percebemos que a primeira (preocupação com o *cache*) leva o desenvolvedor do algoritmo a restringir o mesmo para uma determinada arquitetura de computador.

Nos concentramos em estudar o *QS-5i* por ser um *Quicksort* aleatório e penta-independente e, como entusiasmo adicional, tivemos a idéia de realizar uma análise experimental em um *Quicksort* aleatório, tal análise não fora encontrada em nenhum artigo correlato durante a pesquisa que realizamos.

Expomos também uma análise assintótica mostrando que o *QS-5i* é, aparentemente, tão bom

quanto o *Quicksort* por congruência linear $\left(\mathrm{O}(n \log n)\right)$, porém não apresenta o problema de ter casos com complexidade $\Omega\left(\frac{n^4}{m^2}\right)$.

Apesar de assintoticamente isso ser verdade, confirmamos que, na prática o *QS-5i* se saiu pior do que o *QS-CL* em relação ao tempo real de resposta e número de instruções, os dois algoritmos se comportaram, na média, de forma igual em relação ao número de nodos. O QS-5i só foi melhor (com diferença não muito significativa) no critério trocas.

Considerando todos os resultados médios obtidos na análise experimental, concluímos que o QuickSort-5i é pior do que o QuickSort por congruência linear, e o fato dele não apresentar o "problema" de ter casos com complexidade $\Omega\left(\frac{n^4}{m^2}\right)$ não justifica o seu uso.

Por fim realizamos uma análise experimental comparativa entre 5 algoritmos:

- *Quicksort* com mediana-de-5,
- *Quicksort* versão da GNU (mediana-de-3),
- *Quicksort* 5i ou Penta-Independente,
- *Quicksort* com congruência linear,
- *Quicksort* com pivô fixo na primeira posição,

e seis tamanhos diferentes de n (tamanho da seqüência de entrada), conforme descrito na seção 4.

Durante a análise experimental, percebemos que a versão implementada pela GNU era mais rápida que as versões aleatórias do *Quicksort*, realizamos então um esforço para programar uma versão mais rápida que a da GNU e conseguimos um algoritmo ligeiramente melhor (para o ambiente testado) usando mediana-de-5 ao invés de mediana-de-3. Porém a diferença não foi significativa.

Realizamos então mais testes organizando "um pouco" as sequências de entrada. Para isso foi desenvolvido um programa que rodava mediana-de-7 a alto custo, apenas para organizar "um pouco" as seqüências de entrada.

Um fato interessante foi que os algoritmos que utilizam escolha de pivô aleatória tiveram um fator de melhora em seu desempenho maior do que os outros algoritmos. Concluímos então que a aleatoriedade da escolha do pivô surte maior efeito quando a entrada não é perfeitamente aleatória.

5.2 Trabalhos Futuros

Sugerimos como estudos futuros as seguintes linhas de pesquisa:

- Implementar um *Quicksort* com mediana-de-cinco onde a amostra, usada para calcular a mediana, é aleatória e penta-independente. Experimentar comparativamente aos algoritmos experimentados neste trabalho;
- Implementar e comparar o *Quicksort* raiz de *n* proposto por McGeoch;
- Verificar o comportamento do *Quicksort* probabilístico com outros GNPAs.

Referências Bibliográficas

[Bac91] Eric Bach. Realistic analysis of some randomized algorithms. *J. Comput. Syst. Sci. 42*, pages 30–53, 1991.

[CLRS02] Tomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein. *Algoritimos - Teoria e Prática*. Tradução da segunda edição americana, Campus, 2002.

[KGP94] Donald Knuth, Ronald Graham, and Oren Patashnik. *Concrete Mathematics - A Foundation for computer science*. Addison-Wesley, 1994.

[Knu00a] Donald Knuth. *The Art of Computer Programming - Seminumerical Algorithms*. Addison-Wesley, 2000.

[Knu00b] Donald Knuth. *The Art of Computer Programming - Sorting and Searching*. Addison-Wesley, 2000.

[KR93] Howard J. Karloff and Prabhakar Raghavan. Randomized algorithms ans pseudorandom numbers. *ACM 0004-5411*, pages 700–454, 1993.

[LL97] Anthony LaMarca and Richard E. Ladner. The influence of caches on the performance of sorting. In *Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms*, pages 370–379. Society for Industrial and Applied Mathematics, 1997.

[McG86a] Catherine C. McGeoch. Experimental analysis of algorithms. *J. Exp. Algorithmics*, 1986.

[McG86b] Catherine C. McGeoch. An experimental study of median-selection in quicksort. *Proceedings of the 24th Annual Allerton Conference on Computing, Conrol, and Communication*, pages 19–28, 1986.

[MR01] Conrado Martínez and Salvador Roura. Optimal sampling strategies in quicksort and quickselect. *SIAM J. on Computing, 31(3)*, pages 683–705, 2001.

[MT95] C. McGeoch and J. D. Tygar. Optimal sampling strategies for quicksort. *Random Structures and Algorithms, Vol. 7, No. 4*, 1995.

[Nom03] Alexandre Noma. Análise experimental de algoritmos de planaridade. *Instituto de Matemática e Estatística (IME) Universidade de São Paulo (USP)*, 2003.

[Sed78] Robert Sedgewick. Implementing quicksort programs. *Commun. ACM*, 21(10):847–857, 1978.

[SF96] Robert Sedgewick and Philippe Flajolet. *An Introduction to the Analysis of Algorithms*. Addison-Wesley, 1996.

[XZK00] Li Xiao, Xiaodong Zhang, and Stefan A Kubricht. Improving memory performance of sorting algorithms. *ACM Journal on Experimental Algorithmics*, pages 1–22, 2000.

Apêndice A

Dados dos testes para distribuições uniformes

As tabelas a seguir contém os dados numéricos utilizados para a contrução de vários gráficos apresentados neste trabalho. Além disso, dados adicionais podem ser encontrados nestas tabelas, tais como, razão da grandeza medida pelo n utilizado e, média dessas razões considerando todos os valores de n testados.

QS_5I

QS-5I

Trocas	Usec	Nodos	Instruções
1302839	299813	74801	5334594
13117677	3482293	750273	68117677
27326052	7455624	1520511	139326052
43795002	11222674	2290270	220795097
61072661	15314355	3002018	306972661
75347540	20167483	3845211	392347540

N
262144
2621440
5242880
7864320
10485760
13107200

Trocas / N	Usec / N	Nodos / N	Intruções / N
4.97	1.14	0.29	20.35
5	1.33	0.29	25.98
5.21	1.42	0.29	26.57
5.57	1.43	0.29	28.08
5.82	1.46	0.29	29.28
5.75	1.54	0.29	29.93

Médias

Trocas	Usec	Nodos	Instruções
5.39	1.39	0.29	26.7

Figura A.1: Estatísticas sobre os dados obtidos com os testes para o QS-5i

QS_CL

QS-CL

Trocas	Usec	Nodos	Instruções
1290000	297826	78032	5598732
13290595	3485791	780273	69290595
28029406	7163366	1557531	146029406
45577000	11061253	2339113	230577101
61952281	14762231	3001482	307052281
76810135	20022933	3723608	389810135

N
262144
2621440
5242880
7864320
10485760
13107200

Trocas / N	Usec / N	Nodos / N	Intruções / N
4.92	1.14	0.3	21.36
5.07	1.33	0.3	26.43
5.35	1.37	0.3	27.85
5.8	1.41	0.3	29.32
5.91	1.41	0.29	29.28
5.86	1.53	0.28	29.74

Médias

Trocas	Usec	Nodos	Instruções
5.48	1.36	0.29	27.33

Figura A.2: Estatísticas sobre os dados obtidos com os testes para o QS-CL

QS_GNU

QS-GNU

Trocas	Usec	Nodos	Instruções
1010500	260921	72124	5110587
12106152	29512214	722303	61106152
25716062	6205175	1496572	126716062
41072705	9582019	2167077	206072705
55976829	13019211	2888027	275976829
68837543	16194952	3586672	342837543

N
262144
2621440
5242880
7864320
10485760
13107200

Trocas / N	Usec / N	Nodos / N	Intruções / N
3.85	1	0.28	19.5
4.62	11.26	0.28	23.31
4.9	1.18	0.29	24.17
5.22	1.22	0.28	26.2
5.34	1.24	0.28	26.32
5.25	1.24	0.27	26.16

Médias

Trocas	Usec	Nodos	Instruções
4.87	2.86	0.28	24.28

Figura A.3: Estatísticas sobre os dados obtidos com os testes para o QS-GNU

QS_med5

QS-med5

Trocas	Usec	Nodos	Instruções
1109355	257620	69205	5282659
12715386	2966990	692547	62715386
26690536	6200579	1385423	131690536
40279151	9440379	2078100	200279151
54189164	12820827	2773159	270189164
68738450	16180044	3468415	342738709

N
262144
2621440
5242880
7864320
10485760
13107200

Trocas / N	Usec / N	Nodos / N	Intruções / N
4.23	0.98	0.26	20.15
4.85	1.13	0.26	23.92
5.09	1.18	0.26	25.12
5.12	1.2	0.26	25.47
5.17	1.22	0.26	25.77
5.24	1.23	0.26	26.15

Médias

Trocas	Usec	Nodos	Instruções
4.95	1.16	0.26	24.43

Figura A.4: Estatísticas sobre os dados obtidos com os testes para o QS-med5

QS_Fixo

QS-Fixo

Trocas	Usec	Nodos	Instruções
1692548	299835	79924	5698742
13575321	3508529	7921245	79265912
29594576	7382680	1728541	146593458
46633345	11716549	2419452	231612074
62845573	15275843	3126913	307129042
78843257	20158313	3802498	390810967

N
262144
2621440
5242880
7864320
10485760
13107200

Trocas / N	Usec / N	Nodos / N	Intruções / N
6.46	1.14	0.3	21.74
5.18	1.34	3.02	30.24
5.64	1.41	0.33	27.96
5.93	1.49	0.31	29.45
5.99	1.46	0.3	29.29
6.02	1.54	0.29	29.82

Médias

Trocas	Usec	Nodos	Instruções
5.87	1.4	0.76	28.08

Figura A.5: Estatísticas sobre os dados obtidos com os testes para o QS-Fixo