

Lab CES12 - 2020

Variações de algoritmos de ordenação

VERSÃO 200423

Bugfix nos testes: antes testava 5x com o mesmo vetor. Agora testa com 5 vetores diferentes. Se alguém usou a versão anterior, tem que gerar os dados novamente.

Objetivo:

Implementar variações de algoritmos de ordenação, realizar comparações e entender os resultados.

Implementações:

Todos os algoritmos de ordenação possuem um segundo parâmetro para calcular estatísticas.

Os campos que vocês **devem** medir e preencher são:

`stats.recursive_calls`, que conta o número de chamadas recursivas.

`stats.depth_recursive_stacks`, a profundidade da pilha de recursão.

Para entender a diferença, pense na árvore de recursão: recursive_calls deve ser o número de nós na árvore, depth_recursive_stacks deve ser a altura da árvore.

Também **podem** preencher o campo `stats.custom1`, que é um campo reserva, disponível caso apareça a necessidade de uma medida adicional.

Os outros campos serão preenchidos pelo próprio testador.

```
void myquicksort_2recursion_medianOf3(std::vector<int> &v
                                     , SortStats &stats);
void myquicksort_1recursion_medianOf3(std::vector<int> &v
                                     , SortStats &stats);
```

Variações de quicksort com uma e duas chamadas recursivas

```
void myquicksort_fixedPivot(std::vector<int> &v, SortStats &stats);
```

Quicksort com pivô em índice fixo (sempre o primeiro elemento. Não queremos usar sempre o elemento do meio, pois seria bom para o caso Almost Ordered - queremos que este seja o "primo pobre").

```
void myradixsort(std::vector<int> &v, SortStats &stats);
```

Radix sort, leia a dica sobre implementar com inteiros

```
void mymergesort_recursive(std::vector<int> &v, SortStats &stats);
```

```
void mymergesort_iterative(std::vector<int> &v, SortStats &stats);
```

As duas implementações de mergesort discutidas na aula

Relatório: Comparações

Para cada uma **apresente gráficos**, compare os valores de tempo de execução e explique o resultado

No caso de algoritmos recursivos, a comparação **deve** levar em conta o número de chamadas recursivas realizadas e a profundidade da pilha (a profundidade da pilha é especialmente importante para comparar variações de quicksort).

Note que o testador já gera um arquivo .csv com os valores da struct de estatísticas, incluindo a medida de tempo de execução e os dois campos que devem ser preenchidos pelo aluno.

Ou seja, na prática o aluno apenas precisa implementar as próprias funções de ordenação (inclusive, preencher os campos indicados na struct de estatísticas).

Tempo de execução QuickSort X MergeSort X RadixSort

Considere apenas as melhores versões de cada algoritmo.

MergeSort: Recursivo x Iterativo

QuickSort com 1 recursão x QuickSort com 2 recursões

Aqui é importante verificar a profundidade da árvore de recursão.

QuickSort com mediana de 3 x Quicksort com pivô fixo para vetores quase ordenados

Relatório: Questão opinativa:

Caso real de opinião de engenheiro: “Pode estar demorando muito por culpa da implementação de quicksort da libc, já que alguns dos nossos dados já vem quase ordenados, fica quadrático”. Certamente é possível, mas é plausível ou esperado que uma biblioteca importante, antiga, muito utilizada e bem testada, implemente quicksort de forma a causar o problema citado? Em outras palavras, você, como chefe de um projeto, aprovaria um quicksort implementado assim em um projeto real, considerando o custo-benefício? Justifique.

Nota:

Testes em “SmallTest” não valem nada (com 10 elementos é apenas para ajudar a debugar)

Testes em STL_Tests não valem nada (afinal, testam a função sort da STL. Estes testes servem para garantir que as funções de testes passam se a ordenação funcionar, ou seja, para diminuir a possibilidade de bug nos testes)

Cada teste em “SortTest” vale 1 ponto.

Cada comparação bem explicada e questão bem respondida, vale 1 ponto.

A soma máxima possível de pontos nos testes vale 5

A soma máxima possível de pontos nas questões vale 5

Há alguns testes com vetores de tamanho fixo 1K que estão comentados. Os testes com vários tamanhos devem ser mais relevantes.

Libs auxiliares

a STL está liberada, exceto usar soluções prontas para resolver diretamente o propósito do exercício (especialmente `std::sort`). Em caso de dúvida, pergunte.

Entrega e lib do professor

Há 3 diretórios: tests, lib, e src

Apenas entregue zip *do seu diretório src*.

Compilarei o seu diretório src com a minha cópia do código em tests e lib.

Assim, se mudar algum código em tests ou lib, **isso não será transferido para mim**.

Vale utilizar, se inspirar ou copiar trechos do código fornecido, **mas se quiser mudar algo em lib e src, deve herdar ou criar uma versão sua em src, mudando o nome dos identificadores para evitar conflitos de compilação**. Havendo dúvida pergunte.

Aliás, recomendo que tentem entender o código fornecido, seria um bom aprendizado de OO e testes.

Dicas:

Executar apenas alguns testes:

```
lgm@lgmita:~/labSort/build$ ./labSorttests --gtest_filter=Small*
```

Opção do gtest permite filtrar os testes a ser executados. O exemplo acima, executa apenas os testes com Small no nome, ou seja, apenas os testes com vetor de 10 elementos.

Sugiro que comecem assim, para verificar que o algoritmo funciona, e só depois, executem sem a opção para executar os testes com 1000 elementos e contar o tempo. A vantagem: a tela não é poluída com a saída de erro dos testes maiores (que mostra exemplos de ordenação errada com tamanho 1000), e fica fácil verificar exatamente qual foi o vetor que causou falha no seu algoritmo.

(a opção --help lista todas as opções disponíveis no executável. Essas opções são definidas pelo gtest, não no meu código)

Radix sort com inteiros?

Primeiro, note:

1. Não faz sentido comparar o Radix sort sem uma implementação decente, pois naturalmente ele já tem constantes maiores.
2. no nosso exemplo o valor máximo de cada elemento é o tamanho do vetor, não consideramos todos os valores possíveis de inteiros até MAXINT.

Poderia-se pensar em implementar o radix sort usando os dígitos decimais.

123, 456, 765, seriam separados primeiro pelos dígitos das unidades, depois pelas dezenas, etc.

Mas isso implica em realizar aritmética inteira para encontrar os valores destes dígitos. E.g.: (123 % 100) resulta em 23, depois (23 / 10) resulta em 2 se a divisão for inteira.

É mais rápido pensar em binário! Sabendo o tamanho máximo do número, temos o número de bits a serem considerados.

Usamos uma máscara binária para separar grupos de bits do número, e.g.:

(valor & 0xF0) separa os bits 7,6,5,4 de valor, ou seja, separa um dígito hexadecimal de valor.

Usando o operador >> 4 para deslocar estes números para a direita 4 casas, temos um valor entre 0 e 15 que podemos usar para indexar 16 filas do radixsort.

Como estas operações serão repetidas muitas vezes, usar >> e && ao invés de % e / deve ser mais rápido.

Note que no exemplo acima, a máscara separa 4 bits. Poderíamos separar apenas 3 bits, e teríamos 8 filas.

UPDATE: Pra deixar mais claro:

O objetivo é separar conjuntos de bits do número sem realizar operações inteiras.

Se quer os 4 bits menos significativos (digamos bits 3,2,1,0), basta fazer (valor & 0x0F)

Depois, para os próximos 4 bits, (7,6,5,4), precisa calcular (valor & 0xF0) - mas como o resultado está nos bits (7,6,5,4), precisa shiftar >> 4 bits pra direita para ter um número entre 0 e F que possa ser usado como índice.

A próxima máscara seria 0xF00, para separar os bits (11,10,9,8), e o resultado precisaria shiftar 8 bits. E assim por diante.

Como a própria máscara pode ser shiftada (<<) para gerar a próxima máscara, e o número de casas que o resultado precisa ser shiftado também segue uma regra simples, um loop simples resolve a separação do número em grupos de bits para indexar as listas do radixsort.

E sabemos quando o loop termina, pois o tamanho máximo dos valores a serem ordenados (e portanto o número de bits necessários para representar quaisquer dos valores) é conhecido.

Não sei se é melhor separar de 4 em 4 bits, de 3 em 3 bits, ou etc. É só a idéia, pra vocês pensarem e implementarem. Mais detalhe que isso só se escrever o código pra vocês.

Funções recursivas separadas, não if-else na recursão

Há várias variações de um algoritmo recursivo a serem implementadas. Compare os pseudos-codigos:

```
funcao_recursiva(entrada, variacao)
If (variacao == "nome da 1a variacao do algoritmo")
    Implementacao da 1a variacao
    funcao_recursiva(entrada, variacao)
elseif (variacao == "nome da 2a variacao do algoritmo")
    Implementacao da 2a variacao
    funcao_recursiva(entrada, variacao)
endif
```

```
funcao_recursiva_variacao1 (entrada, variacao) {
    Implementacao da variacao1
    funcao_recursiva_variacao1 (entrada, variacao)
}
funcao_recursiva_variacao2 (entrada, variacao) {
    Implementacao da variacao2
```

```

        funcao_recursiva_variacao2 (entrada, variacao)
    }

```

O problema da primeira versão, **é o gasto de tempo artificial em um if-else que não existe na prática** quando um dos algoritmos for implementado e **que é executado a cada chamada recursiva**. E o código fica misturado, um mesmo código trata de algoritmos diferentes. Além de ser estranho, a princípio não há razão para os algoritmos existirem juntos, estamos apenas comparando. Já forneci headers diferentes para cada variação, é mais realista deixar as implementações separadas. Partes comuns a ambos os algoritmos podem ser implementadas como funções auxiliares.

Formatos

NOME DE ARQUIVO

<nome_do_algoritmo>_MS_[RAND|AO]_<step_size>_<max_size>

RAND: vetor com permutacao aleatoria de 1 a n

AO: vetor quase ordenado

Serao testados tamanhos de step_size a max_size, variando em passos de tamanho step_size.

FORMATO DE ARQUIVO

O operador que coloca a struct SortStats em stream é este:

```

std::ostream& operator<<(std::ostream& stream,
                        const SortStats& st) {
    if (!st.name.empty()) stream << st.name << ", ";
    stream << std::scientific << st.ms_time << ", " <<
st.number_runs << ", " << std::fixed << st.vector_size << ", " <<
std::scientific << st.recursive_calls << ", " <<
st.depth_recursion_stack << ", " << st.custom1 << std::endl;
    return stream;
}

```

Não estou imprimindo o nome para os testes ManySizes. O que significa que a ordem dos campos na linha é:

TIME, NUMBER_RUNS, VECTOR_SIZE, RECURSIVE_CALLS,
DEPTH_RECURSIVE_STACK, CUSTOM1

Onde

- NUMBER_RUNS é o número de repetições, ou seja, ordenações com exatamente os mesmos parametros.
- TIME, RECURSIVE_CALLS, DEPTH_RECURSIVE_STACK, CUSTOM são média dos valores para NUMBER_RUNS execuções com o mesmo tamanho VECTOR_SIZE.

- CUSTOM1 é reserva e não é setado pelo testador. Se precisar medir algo diferente, podem usar.
- RECURSIVE_CALLS e DEPTH_RECURSIVE_STACK são os que vocês devem setar.

FAQ

Tomar a mediana de 3 em cada recursão ou só uma vez no vetor inicial?

Queremos comparar o quicksort aprimorado versus o quicksort 'primo pobre' com pivo fixo. Para o QS aprimorado, qual seria o sentido de correr mais risco de ficar quadrático a partir do próximo nível da recursão? Se sabemos como resolver o problema, resolvemos até o fim, certo?

Tomar a mediana dos 3 primeiros elementos ou do elemento inicial, do meio e final?

O que pode fazer diferença entre essas duas opções é a distribuição dos dados. Nos exemplos quase ordenados, certamente usar os 3 primeiros elementos é bastante estúpido. Supondo que há chance razoável de valores próximos estarem em posições próximas, então usar início meio e fim deve ser melhor, é a solução mais comum.

O que são “opções”? E para executar todos os testes?

Desde o tempo do DOS e Unix (antes de existir Linux ou windows), a libc já tinha API para parsear “options” ao se executar um arquivo executável, usando uma sintaxe

`./nome_do_arquivo -opcao` (com um ou 2 caracteres '-') como no exemplo acima.

Quando o gtest linka o meu código com a `main()` do gtest, está incluído no código do gtest o parseamento e processamento das opções do gtest - por exemplo, pode-se indicar quais testes serão executados.

O nome do executável sozinho (sem opções) significa a execução default.

Por curiosidade uma das APIs existentes é esta:

https://www.gnu.org/software/libc/manual/html_node/Getopt.html

É só um parseador esperto para os parâmetros (`int argc`, `char **argv`) que existem em toda função `main()` em C.

Quem nunca viu isso, experimente:

```
ls --help
```

O próprio comando `ls` tem um texto listando as suas opções.

Pq o CLion pinta o `#include <gtest/gtest.h>` de vermelho no código?

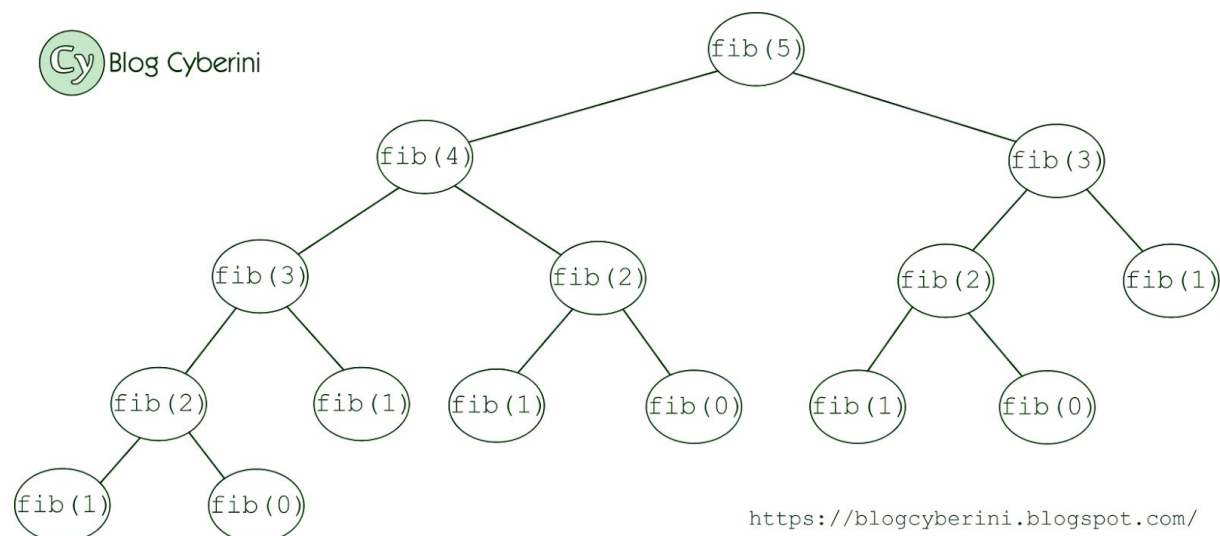
Não usei o CLion, é só um chute, mas pode ser:

Como o gtest esta dentro do directorio build, e o download do gtest é feito durante a propria compilação, pode ser que o editor de texto do CLion (a janelinha e os programas

associados) não saibam onde está o gtest, mesmo que na hora de compilar o cmake encontre o gtest.

Não entendi o que é profundidade da pilha de recursão x número de chamadas recursivas

Exemplo simples de uma árvore de recursão, usando a solução recursiva, simples, para calcular série de fibonacci:



Para resolver fib(5) chama-se fib(4) e fib(3), e assim por diante. O número de chamadas recursivas é o número de nós da árvore (15 no exemplo); e a profundidade da pilha de recursão é a altura máxima da árvore (5 no exemplo). Claro que existem soluções melhores para calcular fibonacci, mas o ponto aqui é só mostrar um exemplo de árvore de recursão.

é permitido adicionar como argumentos das funções de ordenação as posições de início e de fim ? Ou eu preciso manipular por meio de iteradores?

Não pode mudar o cabeçalho, até porque o teste falhará. Também não faz sentido que o usuário de uma lib de ordenação precise enxergar argumentos internos. Crie funções auxiliares com outros cabeçalhos, objetos usados dentro das funções, o que quiser.

Bloopers 2019 (o que não fazer)

Use as minhas funções auxiliares, ou crie as suas. Evite bugs bobos.

Um aluno, ao invés de usar a minha função troca(), implementou manualmente, várias vezes repetindo no código aquelas 3 linhas aux=v[j]; v[j] = v[i]; v[i] = aux; Só que houve um erro no código em uma das trocas, e o pivô do QS foi escolhido errado. O QS ficou sempre quadrático e os dados não faziam sentido.

STL não é mágica, também std::string

A classe std::string, apesar de não ser filha de std::vector, também é implementada internamente como um vetor. Passar um std::string como parâmetro por valor em uma função recursiva, inclui alocar uma nova string como variável local, e alocar espaço para o conteúdo e copiar todo o conteúdo a cada chamada recursiva. Precisa passar como referência.

```
....sort....(std::string s, ....)
```

```
....sort....(std::string &s, ....)
```

A segunda função recursiva pode ser várias vezes mais rápida para o mesmo algoritmo.

Cuidado em adaptar algoritmos existentes. Onde está o pivô?

Por exemplo, se o algoritmo de Quicksort supõe que o pivô está na posição indicada pela variável left, não adianta escolher um pivô em outra posição sem adaptar a idéia.

Por exemplo, se o seu pivô está na posição middle, então precisaria usar troca(v, left, middle) antes de executar o algoritmo do jeito que está, ou adaptar o algoritmo.

Ou seja, não adianta colar código sem entender o que acontece.