Anderson Andrei da Silva, 8944025
Relatório EP3 – 3 Reversão

1.0 BACKTRACKING

Backtracking é um tipo de algoritmo que representa um refinamento da busca por força bruta, em que múltiplas soluções podem ser eliminadas sem serem explicitamente examinadas. O termo foi cunhado pelo matemático estado-unidense D. H. Lehmer na década de 1950.

Uma busca inicial em um programa nessa linguagem segue o padrão busca em profundidade, ou seja, a árvore é percorrida sistematicamente de cima para baixo e da esquerda para direita. Quando essa pesquisa falha, ou é encontrado um nodo terminal da árvore, entra em funcionamento o mecanismo de backtracking. Esse procedimento faz com que o sistema retorne pelo mesmo caminho percorrido com a finalidade de encontrar soluções alternativas.

2.0 BUBBLE SORT

O Bubble Sort, ou ordenação por flutuação (literalmente "por bolha"), é um algoritmo de ordenação dos mais simples. A ideia é percorrer o vector diversas vezes, a cada passagem fazendo flutuar para o topo o maior elemento da sequência. Essa movimentação lembra a forma como as bolhas em um tanque de água procuram seu próprio nível, e disso vem o nome do algoritmo.

No melhor caso, o algoritmo executa operações relevantes, onde n representa o número de elementos do vector. No pior caso, são feitas operações. A complexidade desse algoritmo é de Ordem quadrática. Por isso, ele não é recomendado para programas que precisem de velocidade e operem com quantidade elevada de dados.

3.0 APLICAÇÃO LOCAL

Aqui utilizaremos o backtracking para achar uma sequencia de índices a serem trocados a fim de deixar o vetor recebido ordenado, e o Bubble Sort será adaptado e utilizado em dos casos específicos do problema.

Para facilitar a compreensão trataremos aqui cada troca de valores a partir de um determinado índice como movimento. Como dito à cima, o programa mesmo executa uma busca por força bruta, onde, ao encontrar um primeiro movimento válido repetirá o procedimento até que encontre uma solução ou chegue em um estado em que nada mais pode ser feito. Nesse caso ele retornará e irá desfazer tudo o que a última decisão alterou e então tentará verificar outras possibilidades diferentes. Ao fim, se ainda assim não encontrar nenhuma solução e tiver percorrido todos os “caminhos” possíveis, significa que realmente aquela instância do problema não tem solução.

Ou seja, aqui faremos testes sistemáticos e repetitivos em todas as posições do vetor cuja se encaixe nas devidas condições. Ao encontrarmos um movimento válido o faremos, o que acarretará na mudança do vetor e no início de uma nova busca por mais um movimento válido. E assim prosseguiremos até cairmos em um dos dois casos já descritos.

4.0 IMPLEMENTAÇÃO

Temos um vetor de inteiros de tamanho n > 0 que trataremos de forma “circular”, ou seja, seu fim se conecta com seu início, e dessa forma o vetor não tem fim, apenas volta para o início ao atingir sua posição n.

Para percorrê-lo fazendo os testes necessários utilizaremos duas variáveis auxiliares que terão a função de contadoras. A primeira irá contar a posição, ou seja, será o índice do vetor e será atualizada de 1 em 1, enquanto a segunda controlará o número de tentativas já feitas para aquele vetor. Assim, podemos iniciar a busca de qualquer posição do vetor, e a mesma não se encerrará necessariamente na posição n-1, logo, dado o tamanho do vetor sabemos que ele percorreu todas as posições possível se ele percorreu n posições, assim sendo , a variável que controlará o número de tentativas já percorrida, contabilizará valores de 0 a n sem se preocupar com o real índice atual do vetor e também sendo atualizada de 1 em 1.

Para controlar o backtacking utilizaremos uma pilha de números inteiros de tamanho $2n^2$ onde serão empilhadas as duas variáveis descritas à cima, sendo empilhadas respectivamente a que controla o índice do vetor e depois a que controla o número de tentativas. Dessa forma saberemos qual foi o índice de partida da troca efetuada, e em qual tentativa estávamos, para que então, quando não houverem mais movimentos e então for necessário desempilhar as ultimas decisões tomadas verificarmos se é possível continuar tentando para outras posições e para qual índice devemos desfazer uma troca. E esse tamanho se deve ao fato de que, para cada decisão tomada, 2 números são empilhados, o que pode ser feito n vezes em uma passagem pelo vetor, e que ainda pode ser feito outras n vezes pois o vetor pode ser percorrido n vezes também, assim sendo, teremos 2*(n*n).

4.1 Entrada de dados

O programa recebe um número inteiro n > 0 e um vetor com n inteiros.

4.2 Saída de dados

O programa imprime, caso houver solução, uma série de 3-rotações que ordene este vetor em ordem crescente, sendo representadas pela índice inicial que deve ser trocado. Caso contrário imprime a mensagem “Nao e possivel”.

5.0 DINÂMICA DO ALGORITMO

Trabalhando com vetores de tamanho n, temos dois casos, n par ou n ímpar. Para n ímpar aplicaremos o algoritmo que trabalha com backtracking; e para os casos onde n é par, primeiro faremos testes para verificar se é possível obter uma solução para eles. Em particular para o caso n=2 apenas verificamos se já está ordenado, se não estiver vamos imprimir a mensagem indicando que não é possível ordená-lo, pois não é possível fazer movimento algum com ele.

Nos casos para n par temos a especificidade de que partindo de um índice em específico não é possível chegar em qualquer outro como nos casos de n ímpar. Assim sendo, partindo de um certo índice só “teremos acesso” a outros específicos e então para acessar os que não foram abrangidos por esse “caminho”, temos que usar outro índice de partida.

Para verificar a possibilidade de tal vetor ser ordenado, utilizaremos uma adaptação do algoritmo conhecido como Bubble Sort, e a adaptação se dará nas comparações a serem feitas, que aqui serão entre valores dispostos com a distancia de 1 posição. Dessa forma, serão comparados

todos os pares e todos os ímpares separadamente. Ao final da aplicação dessa adequação do algoritmo teremos que os números pares e os ímpares estarão ordenados separadamente, mas quando intercalados entre si, não se manterão ordenados necessariamente. Assim, se obtivermos um vetor ordenado significa que mesmo com as limitações já especificadas, o menor valor por exemplo, poderá ser posto na primeira posição, assim como o segundo menor na segunda posição e assim respectivamente, permitindo assim serem ordenados. Caso contrário, saberemos que não conseguiremos ordená-los e então nem tentaremos aplicar o algoritmo que com backtracking.

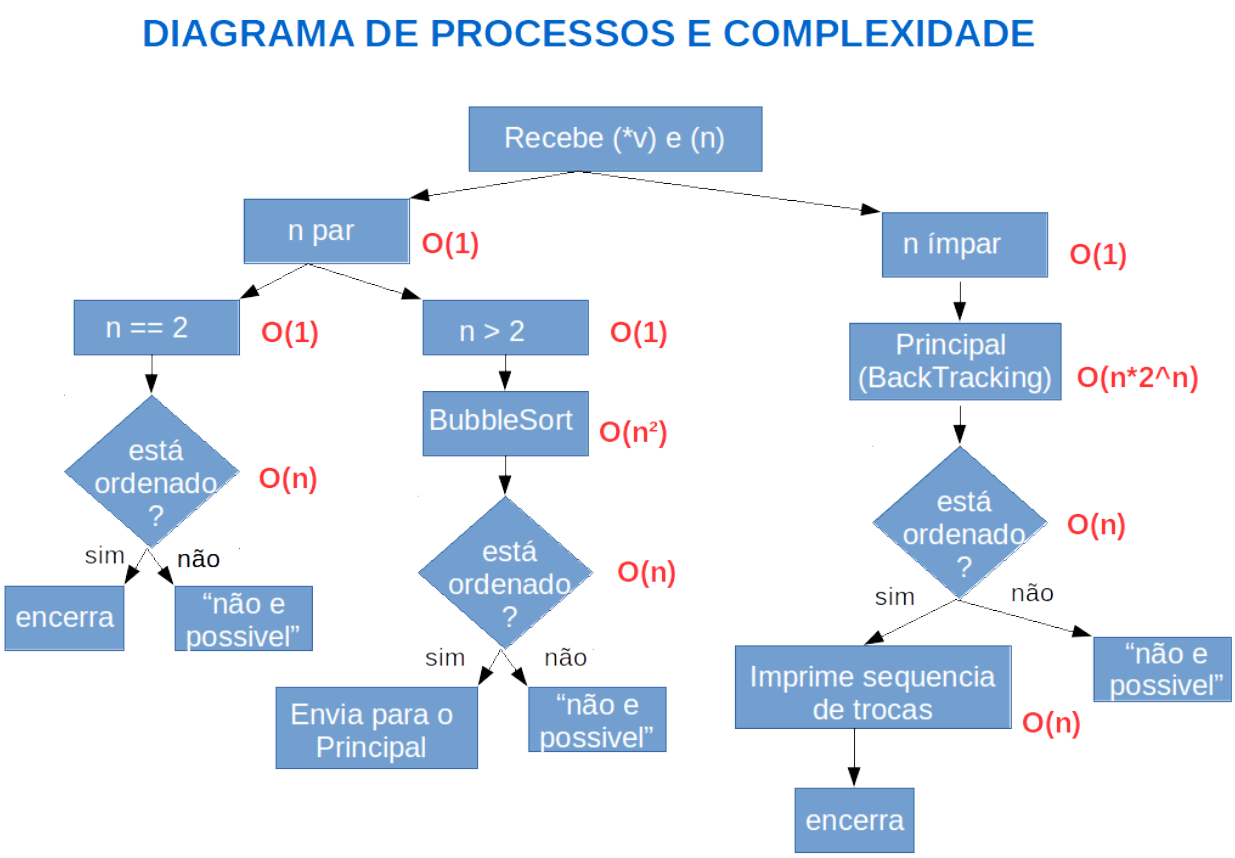
Além dos casos citados acima, vale lembrar que para os casos em que o vetor recebido já estiver ordenado, nada será impresso pois não existem trocas a serem feitas e muito menos motivo para imprimir que não é possível ordená-lo.

Procuramos por 3-Reversões válidas, ou seja, valores que estejam em posições intercaladas por um outro valor qualquer, e que seja o da esquerda menor que o da direita. Quando encontrados eles são trocados de forma que o valor que ficava no meio deles permanece intacto. Em seguida são empilhados o índice da posição do valor que estava à esquerda, e o número ao qual se refere esta tentativa de solução.

Assim, quando não obtemos êxito em um “caminho” , fazemos o bakctracking, desempilhamos os dois últimos valores do topo da pilha, desfazemos a troca e incrementamos o número da tentativa e retomamos o processo.

6.0 ANÁLISE DE COMPLEXIDADE

O algoritmo geral está descrito abaixo juntamente com a complexidade de algumas partes :



Fonte: Edição própria

Podemos observar que os dois processos mais significativos em relação ao consumo de tempo são o Bubble Sort e o processo “Principal”, onde são $O(n^2)$ e O(n*2^n) respectivamente.

O Bubble Sort como descrito acima é um algoritmo que no seu pior caso consome $O(n^2)$ pois faz n comparações n vezes, e no melhor caso pode fazer O(n). Já o nosso processo “Principal” que utiliza o backtracking se comporta da mesma forma pois dado um vetor de tamanho n, dentro das nossas condições, a partir da primeira comparação e consequente escolha de movimento, ele irá fazer outras n tentativas de trocas, e ao encontrar uma situação onde não consegue prosseguir, retrocederá uma decisão (o que efetua mais uma troca) e irá pra outra, ou seja, a partir de 1 primeira decisão, poderão ser tomadas outras n. Como a solução pode não ser encontrada a partir de uma primeira escolha, essa pode ser alterada n vezes também, gerando então, para cada primeira escolha outras n, resultando no pior caso em 2^n trocas da primeira escolha, e n caminhos para cada escolha , o que nos da n*2^n número de comparações, resultando em um consumo O(n*2^n).

No melhor caso, para ambos, seria se o vetor recebido já estiver ordenado. Assim, ambos farão n comparações e nenhuma troca, sendo assim o consumo O(n).

No caso médio, ambos poderão fazer comparações e trocas k vezes, sendo $k < n$, porém ainda sim, no caso do Bubble Sort, ele ainda terá que ser reexecutado outras k vezes, resultando em $k^2 < n^2$; e o nosso processo “Principal” ainda para cada escola terá outras n, ou até mesmo k escolhas, e dobrando cada troca, pensando quando é desfeita uma anterior, em n*2^k ou k*2^k , onde ambos são menores que n*2^n. Ou seja, no caso médio ambos ainda possuem um consumo O(n*2^n).

7.0 FUNÇÕES

Para a implementação do programa, foram utilizadas as seguintes funções:

```
/* Struct que define uma pilha */
typedef struct {
        int topo;
        int *v;
        int tam;
} pilha;

/* Recebe um inteiro n e devolve uma pilha de tamanhao n alocada dinamicamente. */
pilha *criaPilha (int n);

/* Recebe um ponteiro para um pilha e devolve 1 se ela estiver vazia ou 0 caso contrário.*/
int pilhaVazia(pilha *p);

/* Recebe um ponteiro para um pilha e devolve 1 se ela estiver cheia ou 0 caso contrário.*/
int pilhaCheia (pilha *p);

/* Recebe um ponteiro para um pilha, um ponteiro para movimento. Então desempilha o
elemento do topo da pilha e altera diretamente os valores de movimento através do ponteiro
recebido. */
int desempilha (pilha *p);

/* Recebe o ponteiro para um pilha, um ponteiro para movimento e empilha o movimento na
pilha. */
void empilha (pilha *p, int indice);

/* Recebe um ponteiro para pilha e devolve o elemento do topo.*/
int topoPilha(pilha *p);

/* Recebe um ponteiro para um pilha e libera o espaço que foi alocado para a mesma.*/
void destroiPilha(pilha *p);

/* Recebe um ponteiro para um pilha e imprime o que está nela começando do elemento na
posição 0 até o n-1.*/
void imprimePilhaInversa(pilha *p);

/* Recebe um inteiro n e devolve um vetor de tamanhao n alocada dinamicamente. */
int *criaVetor(int n);

/* Recebe um ponteiro para um vetor e um inteiro n representando seu tamanho e atribui 0 a
todas as posições do vetor. */
void zeraVetor (int *v, int n);

/* Recebe um ponteiro para um vetor e libera o espaço que foi alocado para o mesmo. */
void destroiVetor (int *v);

/* Recebe um ponteiro para um vetor e um inteiro n representando seu tamanho e imprime
todos os valores armazenados por ele. */
void imprimeVetor (int *v, int n);
```

/* Recebe um ponteiro para um vetor e um inteiro n representando seu tamanho e checa se ele está ordenado. */
int checaOrdenado (int *v, int n);

/* Recebe um ponteiro para um vetor, dois inteiros n representando seu tamanho e um indice. Ela verifica e retorna qual o indice para uma possível troca. */
int verificaIndiceTroca (int *v, int i, int n);

/* Recebe um ponteiro para um vetor, dois inteiros n representando seu tamanho e um indice. Ela verifica e retorna qual o próximo indice dado um passo. */
int verificaIndicePasso (int *v, int i, int n);

/* Recebe um ponteiro para um vetor, dois inteiros n representando seu tamanho e um indice. Ela verifica e retorna 1 se for possivel efetuar uma troca, ou 0 caso contrário. */
int podeTrocar (int *v, int i, int n);

/* Recebe um ponteiro para um vetor, dois inteiros n representando seu tamanho e um indice e efetua uma troca. */
void troca (int *v, int i, int n);

/* Recebe um ponteiro para um vetor, um inteiro n representando seu tamanho e ordena esse vetor utilizando essensialmente o algoritmo Bubble Sort */
void bubbleSort(int *v, int n);

8.0 QUESTIONAMENTOS

8.1 É possível ordenar um vetor de qualquer tamanho com 3-rotações?

Não. Não é possível ordenar qualquer vetor de qualquer tamanho, mas por conta de sua instância. Vamos aqui separar os números em pares e ímpares e então se formos levar em consideração o tamanho do vetor, podemos ordenar vetores de tamanhos pares e ímpares. Para os casos ímpares podemos ordenar com 3-rotações com qualquer instância, entretanto, para os casos de tamanho par existem instâncias que podem e não podem ser ordenadas para um mesmo n.

8.2 Dado um vetor de tamanho n é possível ordenar qualquer instancia com 3-rotações?

Como dito acima, não. Podem haver casos em que para um determinado número n existe uma instância possível de se ordenar, assim como outras que não. Por exemplo:

Seja n = 4: A instancia [3, 2, 1, 4] é passível de ordenação. Já a instancia [2, 1, 4, 3] não.

8.3 Qual o número máximo de 3-rotações que seu algoritmo executa pra ordenar um vetor?

Como explicado acima, dado um vetor de tamanho n, ele executará no máximo n*2^n 3-rotações.

9. RESULTADOS:

Foi executada uma bateria de testes que retornou que a análise feita está coerente com o que realmente acontece com o algoritmo e para fins de verificar a possibilidade de ordenação serão exibidos alguns exemplos:

1.

Entrada:
1
10
Saída:

real 0m1.944s
user 0m0.004s
sys 0m0.000s

2.

Entrada:
2
40 100
Saída:

real 0m4.136s
user 0m0.000s
sys 0m0.000s

3.

Entrada:
4
3 2 1 4
Saída:
0

real 0m3.328s
user 0m0.000s
sys 0m0.000s

4.

Entrada:
4
2 1 4 3
Saída:
Nao e possivel

real 0m3.096s
user 0m0.000s
sys 0m0.000s

5.

Entrada:
6
6 7 4 2 1 3
Saída:
Nao e possivel

real 0m4.712s
user 0m0.000s
sys 0m0.000s

10.0 REFERÊNCIAS

1.0 BACKTRACKING
https://pt.wikipedia.org/wiki/Backtracking#cite_note-1

2.0 BUBBLE SORT
https://pt.wikipedia.org/wiki/Bubble_sort

4.1 Entrada de dados
4.2 Saída de dados
http://paca.ime.usp.br/pluginfile.php/102308/mod_resource/content/1/ep3.pdf