SCC0201 - Introdução à Ciência da Computação II

Relatório do Projeto 1

Aluno	NUSP
Juan Henriques Passos	15464826

Projeto 1 – Problema da mochila

Força bruta

➔ Comentário

O problema consiste em carregar itens visando maximizar o valor carregado na mochila, com a mochila tendo um espaço máximo, e cada item possui um peso. Uma das formas de resolver esse problema é puxando para cada item duas opções, pegar ou não pegar, e isso vai ocorrer para cada item, dessa forma, passa-se por todas as combinações e garante-se a maior valor, pegando sempre o máximo. Portanto, cada chamada fará mais duas chamadas, pegando ou não pegando o item, e a complexidade disso será O(2^n).

Por árvore de recorrência: a recorrência é definida por T(n) = 2T(n-1), T(n) representa o tempo necessário para resolver um problema de tamanho n. No primeiro nível tem-se apenas uma chamada, no segundo duas, no terceiro 4 e assim por diante. Logo, o número de chamadas é igual a 2^i, sendo i o nível da árvore, até chegar no caso base que é i = 0(altura máximo de n). O número total de nós é 1+2+4+8...+2^(n-1) = (2^n) – 1. E isso corresponde a uma complexidade de O(2^n).

➔ Código

```c
#include<stdio.h>
#include<stdlib.h>
#include<time.h>

int max(int a, int b);
int bruta(int n,int mochila,int *valor,int *espaco);

int main(){
    int n, mochila, *valor, *espaco;

    scanf("%d %d", &n, &mochila);

    valor = (int*) malloc(n * sizeof(int));
    espaco = (int*) malloc(n * sizeof(int));

    for(int i = 0; i < n; i++){
        int a;
        scanf("%d", &a);
        espaco[i] = a;
    }

    for(int i = 0; i < n; i++){
        int a;
```

```c
int main(){
    int n, mochila, *valor, *espaco;

    scanf("%d %d", &n, &mochila);

    valor = (int*) malloc(n * sizeof(int));
    espaco = (int*) malloc(n * sizeof(int));

    for(int i = 0; i < n; i++){
        int a;
        scanf("%d", &a);
        espaco[i] = a;
    }

    for(int i = 0; i < n; i++){
        int a;
        scanf("%d", &a);
        valor[i] = a;
    }

    // Marca o início do tempo
    clock_t start = clock();

    int resposta = bruta(n, mochila, valor, espaco);

    // Marca o fim do tempo
    clock_t end = clock();

    printf("%d\n", resposta);

    // Calcula e imprime o tempo de execução em milissegundos
    double time_spent = (double)(end - start) / CLOCKS_PER_SEC *
    1000;
    printf("Tempo de execução: %f ms\n", time_spent);

    //Desalocação de memoria e boa pratica ao evitar ponteiros
    selvagens.
    free(valor); valor = NULL;
    free(espaco); espaco = NULL;

    return 0;
}

int max(int a, int b){
    if(a > b) return a;
    else return b;
}

int bruta(int n,int mochila,int *valor,int *espaco){
    if(n == 0 || mochila == 0) return 0;

    if(mochila < espaco[n-1]) return bruta(n-1, mochila, valor,
    espaco);

    else return max(
        bruta(n-1, mochila, valor, espaco),
        valor[n-1] + bruta(n-1, mochila - espaco[n-1], valor,
        espaco));
}
```

➔ Saída

Caso teste: quantidade de itens = 30 e capacidade da mochila = 120.

```
83
Tempo de execução: 15.000000 ms
```

Guloso

➔ Comentário

Nessa suposta solução (não está correta pois não garante para todos os casos), faz-se razões entre os valores e os pesos de cada item, dessa forma, pega-se os itens com maior custo-benefício (maior razão) até acabar o espaço na mochila, senão couber, pega-se o próximo.

Em termos de complexidade, o guloso é o melhor, pois como só é necessário ordenar o vetor, no caso foi usado o mergesort que possui complexidade n log n, e após ordenado, percorre-se o vetor apenas uma vez, pegando os itens com maior valor até acabar o espaço na mochila, que no pior dos casos, é n. Logo, a complexidade é O(n log n), por conta que é necessário ordenar o vetor. (Nota-se que um problema no qual o vetor está ordenado, a complexidade é O(n)).

Comprovando que mergesort tem complexidade n log n: a função mergesort consiste em dividir o vetor por dois até possui tamanho 1, e após isso, complementar o vetor novamente, ordenando-os dos casos pequenos como dois elementos, até completar.

Expressão de recorrência: T(n) = 2T(n/2) + O(n)

Divide-se o problema em dois subproblemas com a metade do atual, e em cada nível, o custo de combinar esses subproblemas é igual a n. Como o vetor possui tamanho n e divide-se ele por 2 até possuir tamanho 1:

n/2^k = 1, k é o número de níveis para um vetor de n elementos possui tamanho 1.

k = log n na base 2. E a complexidade é dada por níveis vezes custo de operação em cada nível.

Assim a complexidade do mergesort é O(n log n)

➔ Código

```c
#include<stdio.h>
#include<stdlib.h>
#include<time.h>

typedef struct par{
    float razao;
    int index;
}pair;

int max(float a, float b);
void dividir(pair *vet, int l, int r);
void conquistar(pair *vet, int l, int meio, int r);
int knapsack(int n, int mochila, int *espaco,int *valor);

> int main(){ ···

> int max(float a, float b){ ···

> void dividir(pair *vet, int l, int r){ ···

> void conquistar(pair *vet, int l, int meio, int r){ ···

> int knapsack(int n, int mochila, int *espaco,int *valor){ ···
```

```c
int main(){
    int n, mochila, *valor, *espaco;

    scanf("%d %d", &n, &mochila);

    valor = (int*) malloc(n * sizeof(int));
    espaco = (int*) malloc(n * sizeof(int));

    for(int i = 0; i < n; i++){
        int a;
        scanf("%d", &a);
        espaco[i] = a;
    }

    for(int i = 0; i < n; i++){
        int a;
        scanf("%d", &a);
        valor[i] = a;
    }

     // Marca o início do tempo
    clock_t start = clock();

    int resposta = knapsack(n, mochila, espaco, valor);

    // Marca o fim do tempo
    clock_t end = clock();

    printf("%d\n", resposta);

    // Calcula e imprime o tempo de execução em milissegundos
    double time_spent = (double)(end - start) / CLOCKS_PER_SEC *
    1000;
    printf("Tempo de execução: %f ms\n", time_spent);


    free(valor); valor = NULL;
    free(espaco); valor = NULL;

    return 0;
}

int max(float a, float b){
    if(a > b) return a;
    else return b;
}

void dividir(pair *vet, int l, int r){
    if(l<r){
        int meio = (l+r)/2;

        dividir(vet, l, meio);
        dividir(vet, meio+1, r);

        conquistar(vet, l, meio, r);

    }
}
```

```
void conquistar(pair *vet, int l, int meio, int r){
    int tam1 = meio-l+1;
    int tam2 = r-meio;

    pair L[tam1];
    pair R[tam2];

    for(int i = 0; i < tam1; i++){
        L[i] = vet[i+l];
    }

    for(int i = 0; i < tam2; i++){
        R[i] = vet[i+meio+1];
    }

    int posL = 0, posR = 0, posVet = l;

    while(posL < tam1 && posR < tam2){
        if(L[posL].razao < R[posR].razao){
            vet[posVet] = L[posL];
            posL++;
        }
        else{
            vet[posVet] = R[posR];
            posR++;
        }
        posVet++;
    }

    while(posL < tam1){
        vet[posVet] = L[posL];
        posVet++;
        posL++;
    }
    while(posR < tam2){
        vet[posVet] = R[posR];
        posVet++;
        posR++;
    }
}
```

```c
int knapsack(int n, int mochila, int *espaco,int *valor){
    pair *razao;

    razao = (pair *) malloc(n * sizeof(pair));

    for(int i = 0; i < n; i++){
        razao[i].index = i;
        razao[i].razao = (float)(valor[i])/espaco[i];
    }
    //Ordenar o vetor(n log n) com base na razao.
    dividir(razao, 0, n-1);

    int resposta = 0;
    //Comecar do maior para o menor.
    for(int i = n-1; i >= 0; i--){
        int item = razao[i].index;
        if(espaco[item] <= mochila){
            resposta += valor[item];
            mochila -= espaco[item];
        }
        if(mochila == 0) break;
    }

    free(razao); razao = NULL;

    return resposta;
}
```

➔ **Saída**

Caso teste: quantidade de itens = 1000 e capacidade da mochila = 99999

```
99999
Tempo de execução: 0.047000 ms
```

Observação: um exemplo de caso teste que quebra o algoritmo guloso ->

3 8 (quantidade de itens e capacidade da mochila)
3 4 5 (peso dos itens)
30 50 60 (valor dos itens)

Nota-se que a solução ótima é pegar os itens que pesam 3 e 5, conseguindo 90 de valor. No entanto, as razões desses itens, respectivamente, serão 10 / 12,5 / 12. Dessa forma, o algoritmo guloso terá como resposta 80, pois irá pegar primeiro o item com peso 4, com maior razão, após isso tentara pegar o item 5, mas a mochila so tem 4 de capacidade, sobrando apenas o item 3, totalizando 80 de valor.

Programação dinâmica (DP)

➔ **Comentário**

Nessa solução, segue-se a ideia de testar todos os casos, presente na solução força bruta, porém salvando os valores. Além disso, foi escolhido a implementação da knapsack bottle up, com o objetivo de otimizar memória, ao invés da sua implementação recursiva (top down). A lógica começa nos casos base, pois se temos 0 itens, ou 0 de capacidade não será possível carregar nada, então o valor será zero. Após isso, passa-se por cada item, calculando se dá para pegar o item ou não com aquela capacidade em específica. Formando, uma tabela que guarda o valor levado, que quando chegar no último, que seria com capacidade de entrada da mochila e no enésimo item, tem-se o maior valor possível para aquela capacidade.

Essa implementação é feita com dois laços de repetição, um externo para passar por todos os itens, e outro interno para calcular para aquele item, as capacidades que pode ou não o pegar, salvando na matriz o seu valor. Logo a complexidade é quantidade de itens(n) vezes a capacidade da mochila(w).

Complexidade é O(n*w).

➔ **Código**

```c
#include<stdio.h>
#include<stdlib.h>
#include<time.h>

int **alocar_matriz(int linha, int coluna);
int max(int a, int b);
int knapsack(int n, int mochila,int *valor,int *espaco);

int main(){
    int n, mochila, *valor, *espaco;

    scanf("%d %d", &n, &mochila);

    valor = (int*) malloc(n * sizeof(int));
    espaco = (int*) malloc(n * sizeof(int));

    for(int i = 0; i < n; i++){
        int a;
        scanf("%d", &a);
        espaco[i] = a;
    }

    for(int i = 0; i < n; i++){
        int a;
        scanf("%d", &a);
        valor[i] = a;
    }

     // Marca o início do tempo
    clock_t start = clock();

    int resposta = knapsack(n, mochila, valor, espaco);

    // Marca o fim do tempo
    clock_t end = clock();

    printf("%d\n", resposta);

    // Calcula e imprime o tempo de execução em milissegundos
    double time_spent = (double)(end - start) / CLOCKS_PER_SEC *
    1000;
    printf("Tempo de execução: %f ms\n", time_spent);


    free(valor); valor = NULL;
    free(espaco); valor = NULL;

    return 0;
}

int max(int a, int b){
    if(a > b) return a;
    else return b;
}
```

```c
int **alocar_matriz(int linha, int coluna){
    int **matriz;

    //Ajustando o valor para incluir a posição n.
    linha++; coluna++;

    matriz = (int **) malloc(linha * sizeof(int*));

    matriz[0] = (int*)malloc(linha * coluna * sizeof(int));

    for(int i = 1; i < linha; i++){
        matriz[i] = matriz[0] + coluna * i;
    }

    for(int i = 0; i<linha; i++){
        for(int j = 0; j<coluna; j++){
            matriz[i][j] = 0;
        }
    }

    return matriz;
}

int knapsack(int n, int mochila,int *valor,int *espaco){
    int **dp = alocar_matriz(n, mochila);

    // Percorrer todos os itens.
    for(int i = 1; i <= n; i++){
        // Verificar as melhores soluções com capacidade j para o
        item i.
        for(int j = 0; j <= mochila; j++){
            int peso = espaco[i-1]; //Item começa do 1 e o vetor do
            zero.
            if(peso <= j){
                // Se couber na mochila, a melhor escolha é o maximo
                entre a melhor escolha do item anterior com essa
                capacidade ou o valor do item atual mais a melhor
                escolha no item anterior com essa capacidade menos o
                peso desse item.
                dp[i][j] = max(
                    dp[i-1][j],
                    valor[i-1] + dp[i-1][j - peso]
                );
            }
            else{
                //Se nao couber, nao se pega o item.
                dp[i][j] = dp[i-1][j];
            }
        }
    }
    int resposta = dp[n][mochila];
    //Evitar ponteiros selvagens(boa pratica).
    for(int i = 1; i <= n; i++){
        dp[i] = NULL;
    }
    //Desalocação de memoria.
    free(dp[0]); dp[0] = NULL;
    free(dp); dp = NULL;
    return resposta;
}
```

➔ **Saída**

Caso teste: Número de itens: 1000 Capacidade da mochila: 100000

```
263556
Tempo de execução: 578.000000 ms
```

Obs: 1000ms = 1 segundo