PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL
BACHARELADO EM ENGENHARIA DE SOFTWARE

MARIA EDUARDA MAIA, MARINA YAMAGUTI, SOFIA BATISTA SARTORI

Projeto e Otimização de Algoritmos:

Trabalho 2

PORTO ALEGRE

1 de Julho, 2024

Problema 1:

1. O Problema
   Você está gerenciando uma equipe de desenvolvedores e, a cada semana, precisa escolher entre tarefas de baixa ou alta dificuldade. Tarefas de baixa dificuldade geram uma receita L*i* na semana iii, enquanto tarefas de alta dificuldade geram uma receita H*i*, mas exigem uma semana de preparação sem trabalho na semana anterior. O objetivo é maximizar a receita total ao longo de nnn semanas.

2. Resposta do Item 1
   O algoritmo fornecido tenta resolver o problema, mas pode não retornar a solução ótima em todos os casos.

```
For iterations i = 1 to n
        If hi+1 > li + li+1 then
                Output "Choose no job in week i"
                Output "Choose a high-stress job in week i + 1"
                Continue with iteration i + 2
        Else
                Output "Choose a low-stress job in week i"
                Continue with iteration i +1
        Endif
End
```

Vamos considerar a seguinte instância do problema:

Semana	l	h
1	10	50
2	1	1
3	10	50
4	10	1
5	10	50

Ao Aplicar o algoritmo fornecido:

Iteração 1:

h2 + 1 = 1 + 1 = 2 (não é maior que l1 + l2 = 10 + 1 = 11), então escolhemos um trabalho de baixa dificuldade na semana 1.

Iteração 2:

h3 = 50 (é maior que l2 + l3 = 1 + 10 = 11), então escolhemos "fazer nada" na semana 2 e um trabalho de alta dificuldade na semana 3.

Iteração 4:

h5 = 50 (é maior que l4 + l5 = 10 + 10 = 20), então escolhemos "fazer nada" na semana 4 e um trabalho de alta dificuldade na semana 5.

Resultado do algoritmo:

- Semana 1: Baixa dificuldade (10 dinheiros)
- Semana 2: Fazer nada (0 dinheiros)
- Semana 3: Alta dificuldade (50 dinheiros)
- Semana 4: Fazer nada (0 dinheiros)
- Semana 5: Alta dificuldade (50 dinheiros)

Valor total: 10 + 0 + 50 + 0 + 50 = 110 dinheiros.

A resposta correta, no entanto, seria escolher:

- Semana 1: Alta dificuldade (50 dinheiros)
- Semana 2: Fazer nada (0 dinheiros)
- Semana 3: Alta dificuldade (50 dinheiros)
- Semana 4: Fazer nada (0 dinheiros)
- Semana 5: Alta dificuldade (50 dinheiros)

Valor total: 50 + 0 + 50 + 0 + 50 = 150 dinheiros.

Assim, o algoritmo fornecido não resolve o problema corretamente.

3. Resposta do Item 2

a. O Algoritmo

Para resolver o problema, optamos por uma abordagem de programação dinâmica, que é adequada para problemas de otimização com subestrutura ótima e sobreposição de subproblemas.

O algoritmo envolve criar uma tabela dp onde dp[i] representa a receita máxima até a semana i, levando em consideração as possíveis decisões. Usando a tabela dp, podemos rastrear as decisões feitas para obter a receita máxima. A seguir está o pseudocódigo do algoritmo proposto.

```
Função solveP1(l[], h[]):
        n = tamanho de l[ ] (ou h[ ])
        Se n == 0:
                Retornar 0

        dp[] = vetor de inteiros de tamanho n + 1 inicializado com 0
        task[] = vetor de inteiros de tamanho n + 1 inicializado com 0

        // Caso base para a primeira semana
        dp[1] = max(l[0], h[0])
        task[1] = (h[0] > l[0]) ? 1 : 0
```

```
Para i de 2 até n:
// Escolha a tarefa de baixa dificuldade
Se dp[i - 1] + l[i - 1] > dp[i]:
        dp[i] = dp[i - 1] + l[i - 1]
        task[i] = 0

// Escolha a tarefa de alta dificuldade, se possível
Se i > 1 e dp[i - 2] + h[i - 1] > dp[i]:
        dp[i] = dp[i - 2] + h[i - 1]
        task[i] = 1
        task[i - 1] = -1  // Marque a semana anterior como "não fazer nada"

// Imprimir as tarefas escolhidas
Para i de 1 até n:
        Se task[i] == 0:
                Imprimir "Semana " + i + ": Tarefa de baixa dificuldade"
        Senão Se task[i] == 1:
                Imprimir "Semana " + i + ": Tarefa de alta dificuldade"
        Senão:
                Imprimir "Semana " + i + ": Não fazer nada"
// Imprimir a receita máxima obtida
Imprimir "Receita máxima: " + dp[n]
```

Como mencionado, o algoritmo baseia-se na abordagem de programação dinâmica, sendo assim, é possível definir dois pontos chave para provar sua funcionalidade, a subestrutura ótima e a corretude das escolhas.

1. Subestrutura Ótima:

dp[i] representa a receita máxima até a semana i, considerando as decisões possíveis para aquela semana.

Para cada semana i, temos duas escolhas:

- Escolher uma tarefa de baixa dificuldade e adicionar sua receita a dp[i-1].
- Se a semana i-1 foi marcada como "não fazer nada", então podemos escolher uma tarefa de alta dificuldade e adicionar sua receita a dp[i-2].

Assim, dp[i] é o máximo entre dp[i-1] + l[i-1] e dp[i-2] + h[i-1], dependendo das condições de disponibilidade de semanas.

2. Corretude das Escolhas:

Para provar que o algoritmo proposto funciona corretamente para todas as n semanas, usaremos a técnica de indução matemática.

Base da Indução:

Para i = 1

O algoritmo inicializa dp[1] como o máximo entre l[0] e h[0].

Esta escolha é correta porque, na primeira semana, podemos escolher livremente entre uma tarefa de baixa dificuldade (l[0]) ou uma tarefa de alta dificuldade (h[0]) sem nenhuma restrição prévia.

Portanto, a inicialização para dp[1] = max(l[0], h[0]) está correta.

Passe de Indução:

Suponhamos que o algoritmo funcione corretamente para todas as semanas até i (hipótese de indução). Precisamos mostrar que ele também funciona corretamente para a semana i+1.

Caso 1: Escolhendo uma tarefa de baixa dificuldade na semana i+1

Se escolhermos uma tarefa de baixa dificuldade na semana i+1, a receita total será dp[i]+l[i].

Isso é porque, ao escolher uma tarefa de baixa dificuldade na semana i+1, não há restrições em relação à tarefa escolhida na semana iii.

Portanto, dp[i+1]= dp[i] + l[i]

Caso 2: Escolhendo uma tarefa de alta dificuldade na semana i+1

Para escolher uma tarefa de alta dificuldade na semana i+1, a semana i deve ser uma semana de "não fazer nada".

Portanto, a receita total ao escolher uma tarefa de alta dificuldade na semana i+1 será dp [i−1] + h[i].

Isso é porque estamos somando a receita máxima até a semana i −1 (sem considerar a semana i) com a receita da tarefa de alta dificuldade na semana i +1.

Decisão

Para a semana i + 1, devemos escolher a opção que maximiza dp[i+1].

dp[i+1] = max(dp[i]+l[i], dp[i−1]+h[i])

Esta fórmula assegura que estamos considerando ambas as opções (baixa dificuldade e alta dificuldade) e escolhendo a que proporciona a maior receita acumulada.

Atualização do vetor de tarefas

Atualizamos o vetor task[] para refletir a escolha feita:

- Se escolhemos uma tarefa de baixa dificuldade, task[i+1] é definido como 0.
- Se escolhemos uma tarefa de alta dificuldade, task[i+1] é definido como 1, e task[i] é definido como -1 para marcar a semana anterior como "não fazer nada".

Conclusão da Indução

Por indução, mostramos que se o algoritmo funciona corretamente até a semana i, ele também funcionará corretamente para a semana i+1. Com a base da indução validada para i = 1, podemos concluir que o algoritmo funciona corretamente para todas as semanas de 1 até n. Portanto, o algoritmo proposto garante uma solução ótima para o problema, respeitando as condições específicas de escolha de tarefas de alta dificuldade e maximizando a receita total ao longo das semanas.

b. Análise do Algoritmo

A complexidade temporal do algoritmo é O(n), onde n é o número de semanas. Isto é porque estamos apenas fazendo uma única varredura através do array l[] e h[], calculando valores e fazendo comparações simples, o que leva tempo linear.

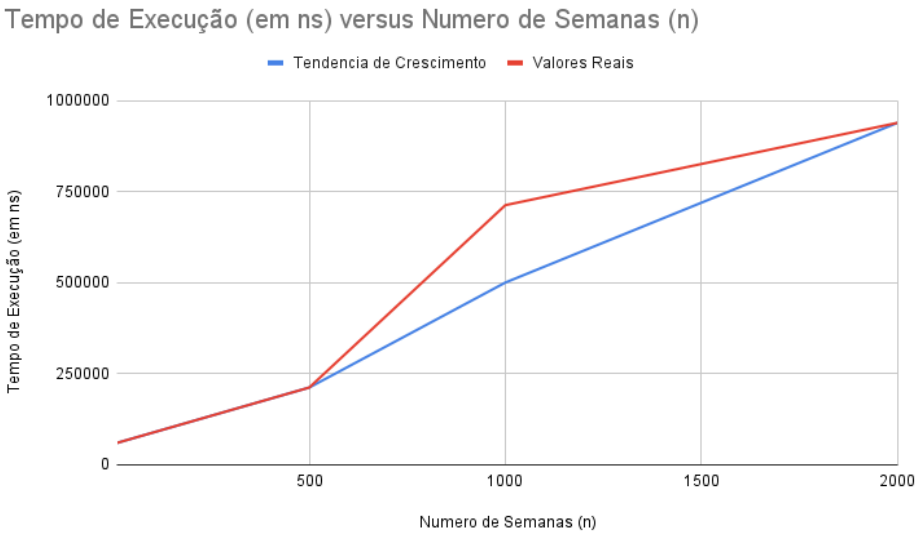
Análise da Complexidade

Inicialização: O tempo de inicialização dos arrays dp e task é O(n).

Iteração: O loop principal percorre de 2 até n e realiza operações constantes dentro do loop, levando O(n) no total.

Para testar o algoritmo, usamos a implementação e os dados de contagem de tempo de execução que serão abordados no tópico 'c'. Com isso, montamos um gráfico (Figura 1) que demonstra que os valores captados no teste de execução realmente seguem uma tendência linear de crescimento.

Figura 1: Tempo de Execução versus Número de Semanas



c. Implementação e Tempo de Execução

Para a implementação do algoritmo proposto utilizamos a linguagem Java, com isso chegamos no seguinte código:

Figura 2: Implementação em Java

```java
public class TaskPlanning {
    public void solveP1(int[] l, int[] h) {
        int n = l.length;
        int[] dp = new int[n + 1];
        int[] task = new int[n + 1];

        // Caso base para a primeira semana
        if (n > 0) {
            dp[1] = Math.max(l[0], h[0]);
            task[1] = (h[0] > l[0]) ? 1 : 0;
        }

        for (int i = 2; i <= n; i++) {
            // Escolha a tarefa de baixa dificuldade
            if (dp[i - 1] + l[i - 1] > dp[i]) {
                dp[i] = dp[i - 1] + l[i - 1];
                task[i] = 0;
            }

            // Escolha a tarefa de alta dificuldade, se possível
            if (i > 1 && dp[i - 2] + h[i - 1] > dp[i]) {
                dp[i] = dp[i - 2] + h[i - 1];
                task[i] = 1;
                task[i - 1] = -1; // Marque a semana anterior como "não fazer nada"
            }
        }

        System.out.println(x:"Plan: ");
        for (int i = 1; i <= n; i++) {
            if (task[i] == 0) {
                System.out.println("Week " + i + ": Low-stress job");
            } else if (task[i] == 1) {
                System.out.println("Week " + i + ": High-stress job");
            } else if (task[i] == -1) {
                System.out.println("Week " + i + ": Do nothing");
            }
        }
        System.out.println("Maximum revenue: " + dp[n]);
    }
}
```

Para testar a execução do algoritmo e marcar seu tempo de execução utilizamos o seguintes método main:

Figura 3: Método Main

```java
public class Main {
    Run | Debug
    public static void main(String[] args) {
        int[] sizes = {1, 10, 500, 1000, 2000, };
        int maxValue = 100;

        for (int size : sizes) {
            int[] l = generateRandomArray(size, maxValue);
            int[] h = generateRandomArray(size, maxValue);

            TaskPlanning p1 = new TaskPlanning();

            long startTime = System.nanoTime();
            p1.solveP1(l, h);
            long endTime = System.nanoTime();

            long duration = (endTime - startTime);
            System.out.println("Execution time for size " + size + ": " + duration + " ns\n");
        }
    }

    public static int[] generateRandomArray(int size, int maxValue) {
        Random random = new Random();
        int[] array = new int[size];
        for (int i = 0; i < size; i++) {
            array[i] = random.nextInt(maxValue) + 1; // Valores entre 1 e maxValue
        }
        return array;
    }
}
```

A tabela a seguir mostra os valores em nanossegundos captados no teste de tempo de execução rodar o método main

Número de Semanas (n)	Tempo de Execução (em ns)
10	59542
500	211625
1000	712583
2000	938750

Problema 2:

1. O Problema

   O problema da mochila (Knapsack Problem) é um problema clássico de otimização combinatória. Dado um conjunto de itens, cada um com um peso e um valor, determine o número de cada item a ser incluído em uma coleção de modo que o peso total não exceda uma capacidade máxima e o valor total seja o maior possível.

   Capacidade da Mochila: 11 kg
   Itens Disponíveis:

- Valor: 18, Peso: 5 kg
- Valor: 22, Peso: 6 kg
- Valor: 6, Peso: 2 kg
- Valor: 1, Peso: 1 kg
- Valor: 28, Peso: 7 kg

2. O Algoritmo

   Branch-and-Bound é uma técnica de busca para resolver problemas de otimização. A ideia é dividir o problema em subproblemas (ramificação) e usar limites para podar soluções que não podem ser ótimas (limitação).

   a. Retorna o valor ótimo para colocar na mochila

   O algoritmo explora todas as possíveis soluções (incluindo ou excluindo cada item) e emprega um limite superior para determinar a melhor solução possível a cada passo. O valor ótimo encontrado representa o maior lucro que pode ser alcançado sem exceder a capacidade da mochila.

   b. Lista quais itens foram colocados na mochila

   Além de calcular o valor ótimo, o algoritmo mantém o controle dos itens que foram incluídos na mochila e que resultam no valor máximo encontrado. Ao final da execução, uma lista dos itens incluídos é exibida.

   Com isso em mente, nós montamos o seguinte algoritmo:

```
Função solveP2()
        Inicializar uma fila Q de nós vazia
        Criar um nó u vazio
        Criar um nó v vazio
        Inicializar v com:
                nível = -1
                lucro = 0
                peso = 0
        Adicionar v à fila Q
        Inicializar maxProfit com 0
        Inicializar bestItems como uma lista vazia
        Enquanto Q não estiver vazia
                Remover o primeiro nó da fila Q e atribuir a v
```

```
                Se v.nível == -1
                        u.nível = 0
                Se v.nível == n - 1
                        Continuar para a próxima iteração do loop
                // Explorar o nó que inclui o próximo item
                u.nível = v.nível + 1
                u.peso = v.peso + wi[u.nível]
                u.lucro = v.lucro + vi[u.nível]
                u.itensIncluídos = copiar lista de itens de v.itensIncluídos
                Adicionar u.nível à lista de itens incluídos de u
                // Verificar se o novo nó é uma solução melhor
                Se u.peso <= W e u.lucro > maxProfit
                        maxProfit = u.lucro
                        bestItems = copiar lista de itens incluídos de u
                // Calcular o bound e adicionar à fila se apropriado
                u.bound = bound(u)
                Se u.bound > maxProfit
                        Adicionar u à fila Q
                // Explorar o nó que não inclui o próximo item
                Criar um novo nó u vazio
                u.peso = v.peso
                u.lucro = v.lucro
                u.nível = v.nível + 1
                u.itensIncluídos = copiar lista de itens de v.itensIncluídos
                u.bound = bound(u)
                Se u.bound > maxProfit
                        Adicionar u à fila Q
        // Exibir resultados
        Imprimir "| - Valor ótimo -> " + maxProfit
        Imprimir "| - Itens colocados na mochila -> "
        Para cada item i em bestItems
                Imprimir i + "  "
                Imprimir nova linha
Fim da função
```

3. Análise do Algoritmo

a. Complexidade de Tempo:

O problema da mochila (Knapsack) é um problema NP-completo. Isso implica que, na pior das hipóteses, o tempo de execução pode ser exponencial em relação ao número de itens n. Além disso, o cálculo do limite superior é normalmente feito de forma linear, adicionando itens até que a capacidade da mochila seja alcançada ou os itens acabem. No pior caso, o algoritmo pode explorar todos os 2^n subconjuntos possíveis dos itens, pois cada item pode ser incluído ou não na mochila. Portanto, a complexidade de tempo na pior das hipóteses é $O(2^n)$

A poda eficiente de ramos não promissores da árvore de busca pode reduzir significativamente o número de nós explorados. O limite superior (bound) ajuda a evitar explorar muitos ramos, especialmente quando eles não podem melhorar a solução ótima encontrada até o momento. No entanto, mesmo com poda, o número de nós explorados ainda pode crescer exponencialmente no pior caso, mas a poda geralmente melhora a performance prática em relação a uma busca exaustiva simples.

Na instância específica do problema que estamos abordando, há 5 itens, então no pior caso, o algoritmo pode explorar 2^5 = 32 subconjuntos possíveis dos itens. Portanto, a complexidade de tempo na pior das hipóteses é O(2^5), que é O(32).

O algoritmo Branch-and-Bound é eficaz em podar grandes partes da árvore de busca, especialmente quando os limites superiores são bem calculados. No entanto, para instâncias de problema grandes, a complexidade ainda pode ser proibitiva, refletindo a dificuldade intrínseca dos problemas NP-completos.

4. Implementação e Tempo de Execução;

   A implementação do algoritmo em Java está estruturada em três classes:

- **Node**: Representa um nó na árvore de decisão, contendo informações sobre o nível, lucro, peso, e itens incluídos.

Figura 4: Classe Node

```java
package Algoritmo2;

import java.util.ArrayList;
import java.util.List;

//--- nó na árvore de decisão -------------------
public class Node {  6 usages  DudaWendelMaia
    int level, profit, bound, weight;  12 usages
    List<Integer> itemsIncluded;  7 usages

    public Node() {  3 usages  DudaWendelMaia
        itemsIncluded = new ArrayList<>();
    }
}
```

- **Backpack**: Contém a lógica do algoritmo Branch-and-Bound para resolver o problema da mochila.

  Métodos Principais:

  - `solveP2()`: Implementa o algoritmo Branch-and-Bound, explora os nós da árvore de decisão, calcula o lucro máximo e os itens que devem ser incluídos na mochila.

Figura 5: Método solveP2 parte 1

```
//--- Resolve o problema e exibe a solução ótima --------
public void solveP2() {  1 usage   DudaWendelMaia
    Queue<Node> Q = new LinkedList<>();
    Node u = new Node(), v = new Node();

    v.level = -1;
    v.profit = v.weight = 0;
    Q.add(v);

    int maxProfit = 0;
    List<Integer> bestItems = new ArrayList<>();

    while (!Q.isEmpty()) {
        v = Q.poll();

        if (v.level == -1)
            u.level = 0;

        if (v.level == n - 1)
            continue;

        //--- Ver o nó que inclui o próximo item ------------
        u.level = v.level + 1;
        u.weight = v.weight + wi[u.level];
        u.profit = v.profit + vi[u.level];
        u.itemsIncluded = new ArrayList<>(v.itemsIncluded);
        u.itemsIncluded.add(u.level);

        //--- Ve se o novo nó é uma solução melhor ----------
        if (u.weight <= W && u.profit > maxProfit) {
            maxProfit = u.profit;
```

Figura 6: Método solveP2 parte 2

```
            bestItems = new ArrayList<>(u.itemsIncluded);
        }

        //--- calcula o limite superior e dependendo add na fila ----
        u.bound = bound(u);

        if (u.bound > maxProfit)
            Q.add(u);

        //--- Ve o nó que não inclui o próximo ---------------
        u = new Node();
        u.weight = v.weight;
        u.profit = v.profit;
        u.level = v.level + 1;
        u.itemsIncluded = new ArrayList<>(v.itemsIncluded);

        u.bound = bound(u);

        if (u.bound > maxProfit)
            Q.add(u);
    }

    System.out.println("----------------------------------------");
    System.out.println("| - Valor ótimo -> " + maxProfit);
    System.out.println("| - Itens colocados na mochila -> ");
    for (int i : bestItems) {
        System.out.print(i + " ");
    }
    System.out.println();
```

- `bound(Node node)`: Calcula o limite superior do lucro para um nó, ajudando a podar ramos não promissores da árvore de decisão.

Figura 7: Método bound

```
//--- Calcula o limite para um nó ---------------
private int bound(Node u) {  2 usages   DudaWendelMaia
    if (u.weight >= W)
        return 0;

    int profit_bound = u.profit;

    int j = u.level + 1;
    int totweight = u.weight;

    // Adiciona itens até o limite da capacidade
    while ((j < n) && (totweight + wi[j] <= W)) {
        totweight += wi[j];
        profit_bound += vi[j];
        j++;
    }

    if (j < n) {
        profit_bound += (W - totweight) * vi[j] / wi[j];
    }

    return profit_bound;
}
```

- **`Main`**: Executa o método `solveP2` e mede o tempo de execução. Ele vai:
    - Definir os dados do problema
    - criar uma instância do solucionador

- executar o método `solveP2`
- medir o tempo de execução

Figura 8: Método Main P2

```java
public static void main(String[] args) {
    //--- Definição dos itens e da capacidade da mochila ---------
    int n = 5;
    int[] wi = {5, 6, 2, 1, 7};
    int[] vi = {18, 22, 6, 1, 28};
    int W = 11;

    Backpack solver = new Backpack(n, wi, vi, W);

    long startTime = System.nanoTime();
    solver.solveP2();
    long endTime = System.nanoTime();

    long duration = (endTime - startTime);  // Calcular a duração
    double durationInSeconds = (double) duration / 1_000_000_000.0; // Converter para segundos

    System.out.println("|- Tempo de execução: " + duration + " nanosegundos (" + durationInSeconds + " segundos)");
    System.out.println("-----------------------------------------------------------------");
```

Ao executar o método Main obtemos os seguintes resultados para o tempo de execução

n	Tempo em Nanossegundos	Tempo em Segundos
5	17636584	0.017636584

Por fim, consideramos que o algoritmo proposto resolve o problema em um tempo de execução eficiente considerando a complexidade do problema.