Análise de desempenho de algoritmos

Data de entrega: 24 de junho de 2024

Grupo: Breno Rosa Almeida, Guilherme Lage da Costa, Marcos Paulo Freitas da Silva e Vinícius Dias

Professor: João Caram Santos de Oliveira

Índice de conteúdos

1. Sobre o problema
2. Introdução e objetivo do estudo
3. Algoritmo *backtracking*
    i. Dados de execução
    ii. Sobre o algoritmo
    iii. Algoritmo implementado
    iv. Massa de testes utilizada
    v. Resultados obtidos
4. Algoritmo guloso
    i. Dados de execução
    ii. Sobre o algoritmo
    iii. Algoritmo implementado
    iv. Massa de testes utilizada
    v. Resultados obtidos
5. Divisão e conquista
    i. Dados de execução
    ii. Sobre o algoritmo
    iii. Algoritmo implementado
    iv. Massa de testes utilizada
    v. Resultados obtidos
6. Algoritmo por programação dinâmica
    i. Dados de execução
    ii. Sobre o algoritmo
    iii. Algoritmo implementado
    iv. Massa de testes utilizada
    v. Resultados obtidos
7. Comparação dos resultados obtidos pelos algoritmos
    i. Tempo de execução
    ii. Lucro encontrado
    iii. Conjuntos solicitados pelo prof. Caram
    iv. Comentários finais sobre os resultados encontrados

Sobre o problema

Uma empresa produtora de energia possui uma quantidade X de energia, medida em megawatts, para vender. Seu objetivo é vender sua energia produzida, obtendo o maior valor possível no conjunto de suas vendas. As vendas serão realizadas por leilão: cada empresa interessada E dará um lance por um lote de K megawatts, oferecendo um valor V por este lote. As interessadas só comprarão um lote do tamanho exato da oferta.

Introdução e objetivo do estudo

Este trabalho visa analisar e comparar a resolução de um mesmo problema a partir de algoritmos distintos, comparando o seu desempenho, ganhos e perdas. Para fins de comparação, foram implementados os seguintes algoritmos:

- Algoritmo *backtracking*;
- Algoritmo guloso;
- Algoritmo de divisão e conquista;
- Algoritmo por programação dinâmica.

Nos tópicos a seguir, serão apresentados os dados de execução de cada um desses algoritmos, bem como os resultados obtidos. Mais adiante, esses dados serão compilados e comparados, concluindo sobre o desempenho de cada um deles.

Algoritmo *backtracking*

Dados de execução

- **Responsável**: Guilherme Lage da Costa
- **Matrícula**: 792939
- **JDK**: Java 17
- **Processador**: AMD Ryzen 5 3600, 4.2 Ghz, 6 cores e 12 threads, 32mb de cache
- **RAM**: 16GB, 3000Ghz
- **Sistema Operacional**: Windows 11
- **IDE**: IntelliJ Ultimate

Sobre o algoritmo

O ***backtracking*** é uma técnica de projetos de algoritmos também conhecida como *retrocesso* ou *tentativa e erro*. Esse algoritmo é um refinamento da busca por força bruta, na qual algumas soluções podem ser descartadas sem que ao menos sejam examinadas. Neste algoritmo, são testadas, metodicamente, várias sequências de decisões até encontrar uma que seja aceitável, ou as descarta até encontrar a melhor solução possível.

De modo geral, esse algoritmo segue inicialmente o padrão de uma busca em profundidade, ou seja, uma árvore é percorrida sistematicamente. Quando essa busca falha, ou é encontrado uma folha da árvore (nós sem filhos), entra em funcionamento o mecanismo que dá nome ao algoritmo de ***backtracking***, fazendo com que o algoritmo retorne pelo mesmo caminho já percorrido, de modo a buscar soluções alternativas que atendam aos critérios pré-definidos.

Algoritmo implementado

```
    public void executar(MelhorResultado melhorResultado, List<Lance> todosLances, List<Lance> lancesS

    int qtdeSelecionada = lancesSelecionados.stream()
            .mapToInt(Lance::quantidade)
            .sum();

    if (lucroAtual > melhorResultado.getLucroMaximizado()) {
        melhorResultado.setLucroMaximizado(lucroAtual);
        melhorResultado.setLancesSelecionados(new ArrayList<>(lancesSelecionados));
    }
    if (indice >= todosLances.size() || qtdeSelecionada >= melhorResultado.getProdutora().quantidadeDi
        return;
    }
    int menorValor = MAX_VALUE;

    for (int i = indice; i < todosLances.size(); i++) {
        menorValor = min(menorValor, todosLances.get(i).quantidade());
    }
    if (qtdeSelecionada + menorValor > melhorResultado.getProdutora().quantidadeDisponivel()) {
        return;
    }
    for (int i = indice; i < todosLances.size(); i++) {
        Lance lance = todosLances.get(i);

        if (qtdeSelecionada + lance.quantidade() < melhorResultado.getProdutora().quantidadeDisponivel
            lancesSelecionados.add(lance);
            executar(melhorResultado, todosLances, lancesSelecionados, i + UM, lucroAtual + lance.valo
            lancesSelecionados.remove(lancesSelecionados.size() - UM);
        }
    }
}
```

No algoritmo implementado, a função responsável por executar o método de ***backtracking*** recebe cinco parâmetros, sendo eles:

- `melhorResultado : MelhorResultado`
- `todosLances : List<Lance>`
- `lancesSelecionados : List<Lance>`
- `indice : int`
- `lucroAtual : int`

O parâmetro `melhorResultado` contém informações sobre a empresa produtora e compradoras, um contador de tempo de execução, a lista dos lances selecionados, o lucro maximizado e a quantidade disponibilizada para venda pela empresa produtora. O parâmetro `todosLances` relaciona todos os lances feitos por todas as empresas compradoras, enquanto `lancesSelecionados` armazena todos os lances selecionados que compõem a combinação em análise. O parâmetro `indice` indica qual o índice da lista de lances `todosLances` que será analisado, de modo a verificar se ele pode ser inserido na combinação atual.

O algoritmo se inicia quantificando a quantidade total dos lances atualmente selecionados (em megawatts) e armazena esse valor na variável `qtdeSelecionada` .

```
int qtdeSelecionada = lancesSelecionados.stream()
        .mapToInt(Lance::quantidade)
        .sum();
```

Se o lucro atual (`lucroAtual`) for maior do que o valor do maior lucro registrado, armazenado no atributo `lucroMaximizado` do `melhorResultado` , o algoritmo atualiza o lucro maximizado e a lista de lances selecionados.

```
if (lucroAtual > melhorResultado.getLucroMaximizado()) {
    melhorResultado.setLucroMaximizado(lucroAtual);
    melhorResultado.setLancesSelecionados(new ArrayList<>(lancesSelecionados));
}
```

O algoritmo verifica se o índice passado como parâmetro é igual ou maior à quantidade total de lances, ou se a quantidade selecionada é maior ou igual à quantidade disponível para leilão pela produtora. Se qualquer dessas condições for verdadeira, a execução termina.

```
if (indice ⩾ todosLances.size() || qtdeSelecionada ⩾ melhorResultado.getProdutora().quantidadeDispor
    return;
}
```

Em seguida, o algoritmo calcula o menor valor de lance a partir do índice atual para garantir que há espaço suficiente para adicionar um novo lance sem exceder a capacidade de venda.

```
int menorValor = MAX_VALUE;

for (int i = indice; i < todosLances.size(); i++) {
    menorValor = min(menorValor, todosLances.get(i).quantidade());
}
if (qtdeSelecionada + menorValor > melhorResultado.getProdutora().quantidadeDisponivel()) {
    return;
}
```

Para cada lance válido a partir do índice atual, o algoritmo adiciona o lance à lista de lances selecionados e chama recursivamente a função `executar` com o próximo índice e o lucro atualizado.

```
for (int i = indice; i < todosLances.size(); i++) {
    Lance lance = todosLances.get(i);

    if (qtdeSelecionada + lance.quantidade() < melhorResultado.getProdutora().quantidadeDisponivel())
        lancesSelecionados.add(lance);
        executar(melhorResultado, todosLances, lancesSelecionados, i + UM, lucroAtual + lance.valor())
        lancesSelecionados.remove(lancesSelecionados.size() - UM);
    }
}
```

De modo geral, o algoritmo de backtracking implementado busca encontrar a combinação de lances que maximiza o lucro total, respeitando a capacidade de venda da empresa produtora. Para melhorar o seu desempenho, foram realizadas podas em três pontos distintos, (i) quando o índice atual é maior do que o índice final da lista `todosLances` ou a quantidade selecionar for superior ao total disponibilizado para leilão pela produtora; (ii) quando o menor valor de lance disponível for maior do que o espaço restante para leilão; e (iii) quando a quantidade selecionada mais a quantidade de um próximo lance for maior do que a disponibilizada, e dessa forma o algoritmo não executa essa iteração. Ele realiza as seguintes etapas:

1. Calcula a quantidade total dos lances selecionados.
2. Atualiza o melhor resultado encontrado, se o lucro atual for maior que o lucro maximizado.
3. Verifica condições de terminação com base no índice e na quantidade selecionada.
4. Calcula o menor valor de lance a partir do índice atual.
5. Itera sobre os lances válidos, adiciona-os à lista de lances selecionados, e chama recursivamente a função `executar`.

Ao final da execução, o algoritmo retorna a melhor combinação de lances encontrada, maximizando o lucro total.

Massa de testes utilizada

A massa de testes utilizada seguiu os seguintes parâmetros:

- **Quantidade *mínima* p/ compradora = 1000** → indica a quantidade mínima que uma determinada compradora poderia solicitar em um lote;
- **Quantidade *máxima* p/ compradora = 1500** → indica a quantidade máxima que uma determinada compradora poderia solicitar em um lote;
- **Quantidade disponível pela produtora = 8000** → indica a quantidade total (lote de megawatts) que a empresa produtora possui, ou seja, que disponibiliza para leilão;
- **Quantidade máxima de lances p/ compradora = 1** → indica a quantidade máxima de lances que cada compradora poderia fazer;
- **Quantidades de compradoras = [10, ..., 33]** → foram executados 10 testes para cada quantidade de lances, iniciado em 10 e incrementado de 1 a 1 até atingir um tamanho em que o problema não foi possível de ser resolvido em até 30 segundos pelo algoritmo. Quando isso aconteceu, foram executados os 10 testes com essa massa e em seguida a execução foi finalizada. Na implementação realizada, o algoritmo conseguiu executar massas de testes com **10 ** até **33** lances. Para cada um desses cenários, foram criados novos conjuntos de testes para que a média de tempo fosse calculada, por exemplo, para executar 27 lances, foram criados 10 listas com 27 lances diferentes e em seguida calculado o tempo médio entre cada uma dessas execuções. Além desses conjuntos de compradoras e lances, gerados aleatoriamente, foram executados dois conjuntos de testes específicos encaminhados pelo prof. Caram, com 25 lances cada um.

Os resultados gerados após cada execução do algoritmo foram armazenados automaticamente em dois arquivos: `exec-log.xls` e `hist-log.xls`. O primeiro log guarda dados gerais da execução como o tempo despendido, algoritmo utilizado, recursos computacionais disponíveis, dentre outros, enquanto o segundo log guarda os dados sobre os lances que foram feitos e os que foram escolhidos para combinar o melhor resultado.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
1	Algoritmo	Data de execucao	Qtde lances	Qtde lances selecionados	Melhor resultado	Tempo execucao (seg)	Cores disponiveis	Memoria total (Mb)	Memoria maxima (Mb)	Memoria liberada (Mb)									
2	BACKTRACKING	14-05-2024 08:54:22	10	6	417, 0	12	268	4278	262										
3	BACKTRACKING	14-05-2024 08:54:22	20	10	667, 0	12	268	4278	262										
4	BACKTRACKING	14-05-2024 08:54:25	30	14	914, 3	12	268	4278	262										
5	BACKTRACKING	14-05-2024 08:55:48	40	14	931	83	12	268	4278	262									
6	BACKTRACKING	14-05-2024 09:38:39	50	20	1152	2571	12	268	4278	262									
7	BACKTRACKING	14-05-2024 09:39:51	10	6	401, 0	12	268	4278	262										
8	BACKTRACKING	14-05-2024 09:39:51	20	13	723, 0	12	268	4278	262										
9	BACKTRACKING	14-05-2024 09:39:53	30	13	841, 2	12	268	4278	262										
10	BACKTRACKING	14-05-2024 09:41:04	40	15	897	71	12	268	4278	262									
11	BACKTRACKING	14-05-2024 13:42:42	10	5	317, 0	12	268	4278	262										
12	BACKTRACKING	14-05-2024 13:42:42	20	14	754, 0	12	268	4278	262										
13	BACKTRACKING	14-05-2024 13:42:53	30	17	1208	11	12	268	4278	262									
14	BACKTRACKING	14-05-2024 13:43:56	40	14	930	63	12	268	4278	262									
15	BACKTRACKING	14-05-2024 18:21:42	10	9	451, 0	12	268	4278	262										
16	BACKTRACKING	14-05-2024 18:21:43	20	14	835, 1	12	268	4278	262										
17	BACKTRACKING	14-05-2024 18:21:45	30	14	936, 2	12	268	4278	262										
18	BACKTRACKING	14-05-2024 18:26:43	40	17	1067	298	12	268	4278	262									
19	BACKTRACKING	14-05-2024 18:28:27	50	14	973	104	12	268	4278	262									
20	BACKTRACKING	14-05-2024 18:54:07	10	7	537, 0	12	268	4278	262										
21	BACKTRACKING	14-05-2024 18:54:07	20	12	610, 0	12	268	4278	262										
22	BACKTRACKING	14-05-2024 18:54:14	30	14	811, 7	12	268	4278	262										
23	BACKTRACKING	14-05-2024 18:55:49	40	17	1030	95	12	268	4278	262									
24	BACKTRACKING	14-05-2024 20:37:19	10	8	470, 0	12	268	4278	262										
25	BACKTRACKING	14-05-2024 20:37:19	20	13	824, 0	12	268	4278	262										

[Exemplo de arquivo de log `exec-log.xls`]

	A	B	C	D	E	F	G	H	I	J	K	L
19	Valor total, 417											
20	Quantidade requisitada	Valor ofertado	Historico									
21	14	86	hist-2									
22	22	1	hist-2									
23	93	12	hist-2									
24	34	63	hist-2									
25	54	63	hist-2									
26	77	92	hist-2									
27	22	53	hist-2									
28	33	31	hist-2									
29	18	55	hist-2									
30	34	37	hist-2									
31	3	64	hist-2									
32	14	34	hist-2									
33	15	91	hist-2									
34	62	46	hist-2									
35	72	91	hist-2									
36	37	40	hist-2									
37	2	29	hist-2									
38	46	79	hist-2									
39	9	53	hist-2									
40	26	94	hist-2									
41	-----	------										

[Exemplo de arquivo de log `hist-log.xls`]

O arquivo de análises que compila as execuções realizadas pode ser verificado no arquivo analise backtracking.

Resultados obtidos

Conforme descrito acima, os cenários de testes variaram de 10 a 33 lances, para cada um desses cenários, foram executados 10 iterações com conjuntos distintos de lances, gerados em tempo de execução, de modo que foram

executadas * *240** execuções ao total (ou seja, (33 - 10) * 10 + 10). Na tabela a seguir, estão relacionados cada um dos cenários de testes (linhas), e o tempo médio despendido em cada uma das suas iterações (colunas). Os casos em que o tempo de execução foi inferior a 1 segundo estão indicados com o símbolo "-", ou seja, o tempo de execução não é significativo.

Quantidade lances	1	2	3	4	5	6	7	8	9	10
10	-	-	-	-	-	-	-	-	-	-
11	-	-	-	-	-	-	-	-	-	-
12	-	-	-	-	-	-	-	-	-	-
13	-	-	-	-	-	-	-	-	-	-
14	-	-	-	-	-	-	-	-	-	-
15	-	-	-	-	-	-	-	-	-	-
16	-	-	-	-	-	-	-	-	-	-
17	-	-	-	-	-	-	-	-	-	-
18	-	-	-	-	-	-	-	-	-	-
19	-	1,00	-	-	-	-	-	-	-	-
20	-	-	-	-	-	-	-	-	1,00	-
21	-	-	-	-	-	-	-	-	1,00	-
22	-	-	-	-	1,00	-	-	-	-	-
23	1,00	-	-	1,00	-	-	-	1,00	-	-
24	1,00	-	-	1,00	-	1,00	-	1,00	-	1,00
25	-	1,00	-	-	1,00	1,00	1,00	1,00	1,00	-
26	1,00	1,00	1,00	1,00	1,00	1,00	2,00	-	1,00	1,00
27	3,00	1,00	2,00	1,00	1,00	1,00	2,00	1,00	-	2,00
28	4,00	3,00	2,00	1,00	1,00	2,00	2,00	3,00	2,00	3,00
29	6,00	4,00	7,00	4,00	6,00	4,00	1,00	12,00	2,00	4,00
30	15,00	9,00	2,00	7,00	11,00	3,00	4,00	16,00	5,00	4,00
31	12,00	6,00	10,00	12,00	7,00	6,00	10,00	12,00	7,00	6,00
32	19,00	12,00	18,00	8,00	11,00	26,00	15,00	5,00	7,00	6,00

Quantidade lances	1	2	3	4	5	6	7	8	9	10
33	27,00	11,00	12,00	11,00	6,00	14,00	34,00	42,00	11,00	21,00

[Tabela de tempos médios de execução - Backtracking]

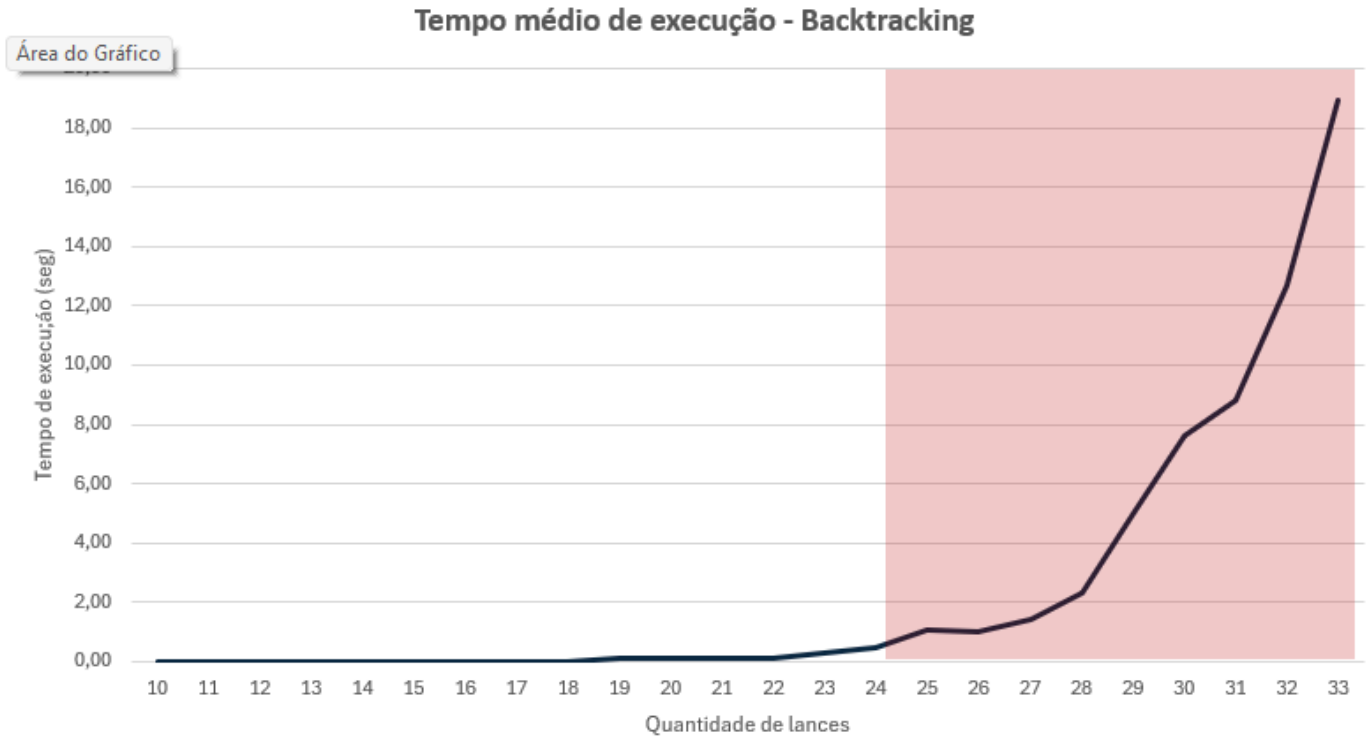
Os valores de lucro máximo obtido em cada um dos cenários de testes, e respectivas iterações, são apresentados na tabela a seguir.

Quantidade lances	1	2	3	4	5	6	7	8	9
10	11.373	8.394	9.690	9.040	10.005	8.368	10.421	10.152	10.625
11	11.045	9.075	12.153	10.521	8.854	9.458	11.361	10.629	10.105
12	11.395	12.706	9.367	10.763	11.212	9.854	12.350	11.432	9.855
13	11.059	9.279	9.575	11.873	9.421	9.821	11.747	12.300	10.670
14	9.274	14.021	10.814	14.062	11.814	11.063	11.309	13.356	12.220
15	13.535	8.200	13.383	13.756	11.483	14.188	13.611	11.064	12.690
16	12.160	13.591	12.249	13.478	11.905	12.510	12.011	12.417	12.924
17	13.424	12.573	14.009	12.169	14.769	13.004	16.050	12.196	12.516
18	14.208	11.297	15.407	11.440	14.408	10.804	15.238	16.301	13.775
19	12.453	14.352	13.737	13.724	13.662	12.215	12.236	14.697	11.784
20	14.831	14.268	15.035	11.828	13.712	13.552	14.871	13.827	13.986
21	11.951	16.218	13.216	13.626	15.327	14.926	13.176	14.002	13.501
22	14.270	13.253	13.694	15.215	13.076	11.853	14.362	14.574	16.733
23	14.160	14.795	15.002	13.760	13.577	16.308	14.030	14.619	15.402
24	14.003	13.505	14.812	15.435	14.425	15.916	15.243	14.105	15.037
25	15.837	14.717	13.223	13.437	15.213	15.876	16.253	17.029	14.090
26	15.217	14.726	16.423	17.017	13.700	14.073	15.667	12.722	15.887
27	16.934	15.958	17.389	14.517	14.770	14.705	16.279	15.783	13.216
28	16.909	15.472	15.263	14.260	13.573	14.049	13.686	17.431	15.361
29	15.956	16.563	16.741	17.101	16.104	14.728	14.132	17.076	15.299

Quantidade lances	1	2	3	4	5	6	7	8	9
30	17.738	17.273	14.179	16.732	16.281	14.321	16.194	18.770	14.250
31	16.662	14.927	16.779	14.237	15.641	14.458	17.013	18.222	14.349
32	17.290	15.307	17.305	15.826	13.945	17.748	14.959	14.481	14.309
33	17.028	16.308	15.647	14.528	16.205	17.425	17.212	18.868	15.911

[Tabela de maior lucro obtido - Backtracking]

Os dados indicados na tabela de tempos médios acima são consolidados no gráfico abaixo, que apresenta a evolução do tempo médio despendido em cada cenário de teste.



[Gráfico dos tempos médios obtidos com o algoritmo de backtracking]

Além dos cenários apresentados, foram executados dois conjuntos adicionais fornecidos pelo prof. Caram, conjunto um e dois. A relação das informações obtidas com a execução desses conjuntos são elencadas na tabela abaixo.

Conjunto	Quantidade lances	Lances selecionados	Lucro máximo	Tempo execução (seg)
Um	25	19	R$ 26.725,00	4
Dois	25	21	R$ 40.348,00	3

[Tabela de resultados para os conjuntos do prof. Caram - Backtracking]

Conforme observado no gráfico apresentado, até a execução com 26 lances, o algoritmo de backtracking conseguia encontrar o melhor lucro do problema em menos de 1 segundo. A partir desse ponto, os tempos de execução passaram a ser maiores que 1 segundo e começaram a crescer exponencialmente. Com apenas 2 lances adicionais (28 lances), o tempo de execução foi de 2,3 segundos, um aumento de 130%. Quando o algoritmo executou o cenário com 33 lances, foi atingido o limite de 30 segundos, fazendo com que esse fosse o último cenário analisado. Neste caso, o tempo médio de execução foi de 18,9 segundos, ou seja, 722% superior ao cenário com 28 lances.

Os resultados obtidos indicam que, em cenários com baixa quantidade de dados, o backtracking pode se mostrar uma opção viável, uma vez que o tempo de execução não será significativo. No entanto, à medida que a quantidade de dados começar a crescer muito, ou uma poda pouco efetiva é utilizada (de modo que o algoritmo execute muitas operações recursivamente, sem conseguir "podar" muitos cenários), o tempo de execução deste algoritmo pode crescer significativamente, deixando de ser uma opção interessante para resolver o problema.

Por se tratar de um refinamento do algoritmo de força bruta, em que algumas das combinações possíveis podem ser descartadas por meio do critério de poda, é de suma importância que esse critério seja bem definido para a execução satisfatória do algoritmo, uma vez que a quantidade de combinações possíveis, dado um conjunto "n", é da ordem de 2^n - 1, ou seja, para o caso de 33 lances, existem 2^33 - 1 combinações possíveis. Conforme apresentado anteriormente, foram aplicados três critérios distintos de poda, de modo que o algoritmo não executasse combinações desnecessárias, e, simultaneamente, conseguisse encontrar o maior lucro possível em um tempo razoável.

Algoritmo guloso

Dados de execução

- **Responsável**: Vinícius Dias
- **Matrícula**: 728272
- **JDK**: Java 17
- **Processador**: 13ª geração Intel Core i5-13450HX (10-core, cache de 20MB, até 4.6GHz)
- **RAM**: Memória de 16GB DDR5 (2x8GB) 4800MHz
- **Sistema Operacional**: Ubuntu 22.04
- **IDE**: IntelliJ Ultimate

Sobre o algoritmo

O algoritmo de `Guloso` é conhecido por ter um ótimo desempenho e não garantir o melhor resultado. Ele é estruturado fazendo a escolha que parece ser a melhor a cada interação. Sendo assim como sabemos qual escolha é a melhor?

Para isso usamos o chamado "Critério Guloso" uma forma de tentar máximizar as boas decisões, mesmo sem garantia que isso pegará o melhor resultado. O critério guloso verifica o conjunto de soluções produzidos pelos candidatos atuais, sem ter uma visão toda do problema, depois seleciona no conjunto o cadidato mais promissor até achar uma solução. Um exemplo disso é o problema da mochila, como critério guloso podemos ordenar as opções pelo maior valor e assim ir selecionando os itens até o peso máximo ser atingido.

Note que assim que o algoritmo encontra uma solução, já é usada como o melhor resultado possível dele, porque ele não vai verificar outras possíveis soluções, o que encurta drasticamente o tempo de execução, mas deixando prá trás outras possíveis soluções que poderiam ter um valor melhor.

Algoritmo implementado

Guloso 1

```
@Override
 public void executar(@NotNull MelhorResultado resultado, List<entidades.Lance> todosLances, List<e

    List<Lance> lancesMutaveis = new ArrayList<>(todosLances);
    lancesMutaveis.sort(comparingDouble(Lance::valorPorMegawatt));

    int energiaRestante = resultado.getProdutora().quantidadeDisponivel();

    for (Lance lance : lancesMutaveis) {
        lancesSelecionados.forEach(lanceSelecionado -> {
            if (Objects.equals(lance.id(), lanceSelecionado.id())) {
                lancesSelecionados.remove(lanceSelecionado.id());
            }
        });
        if (lance.quantidade() <= energiaRestante) {
            lancesSelecionados.add(lance);
            energiaRestante -= lance.quantidade();
        }
    }
    int valorTotal = ZERO;
    resultado.setLancesSelecionados(new ArrayList<>(lancesSelecionados));

    for (Lance lance : lancesSelecionados) {
        valorTotal += lance.valor();
    }
    resultado.setLucroMaximizado(valorTotal);
    resultado.setQuantidadeVendida(resultado.getProdutora().quantidadeDisponivel() - energiaRestar
}
```

Guloso 2

```
@Override
  public void executar(@NotNull MelhorResultado resultado, List<entidades.Lance> todosLances, List<e

    List<Lance> lancesMutaveis = new ArrayList<>(todosLances);
    lancesMutaveis.sort(comparingDouble(Lance::quantidade));

    int energiaRestante = resultado.getProdutora().quantidadeDisponivel();

    for (Lance lance : lancesMutaveis) {
        lancesSelecionados.forEach(lanceSelecionado -> {
            if (Objects.equals(lance.id(), lanceSelecionado.id())) {
                lancesSelecionados.remove(lanceSelecionado.id());
            }
        });
        if (lance.quantidade() <= energiaRestante) {
            lancesSelecionados.add(lance);
            energiaRestante -= lance.quantidade();
        }
    }
```

```
        int valorTotal = ZERO;
        resultado.setLancesSelecionados(new ArrayList<>(lancesSelecionados));

        for (Lance lance : lancesSelecionados) {
            valorTotal += lance.valor();
        }
        resultado.setLucroMaximizado(valorTotal);
        resultado.setQuantidadeVendida(resultado.getProdutora().quantidadeDisponivel() - energiaRestar

    }
```

Método `executar` **:** Este método é o próprio algoritmo em si, que a cada execução vai receber por parâmetro os lances para serem análisados e uma classe chamada "MelhorResultado", para preencher a melhor solução achada pelo algoritmo. Detalhe há dois algoritmos com uma sútil diferença como foi pedido na proposta, dois algoritmos gulosos com critério guloso diferente.

```
List<Lance> lancesMutaveis = new ArrayList<>(todosLances);
```

Cópia Modificável da Lista de Lances:

- `todosLances` é a lista original de lances. No Java, não podemos alterar uma lista como essa, então criamos uma lista mutável que vai receber todos os lances chamada `lancesMutaveis` .

```
lancesMutaveis.sort(comparingDouble(Lance::valorPorMegawatt).reversed());
```

```
lancesMutaveis.sort(comparingDouble(Lance::quantidade));
```

Critério Guloso:

- O critério guloso do primeiro algoritmo, foi pensado de uma forma para deixar os melhores candidatos nas primeiras posições, realizando uma razão do valor pelo megawatt, assim os mais valiosos eram colocados nas primeiras posições
- O critério guloso do segundo alforitmo é muito simples, ordenar em ordem crescente os lances por megawatt, feito para mostrar, qual a diferença se criarmos um critério guloso bem planejado e usar um que não sabemos se trará bons ou ruins resultados.
- Todos os lances que entrarem na solução, irão ser adicionados numa lista chamada `lancesSelecionados`

```
int energiaRestante = resultado.getProdutora().quantidadeDisponivel();
```

Energia Restante:

- É criada uma variável chamada energia restante, que receberá o valor que a produtora de energia tem em megawatts para oferecer. Ela é usada depois para ter um controle da quantidade, decrementando seu valor a cada lance adicionado a solução.

```
for (Lance lance : lancesMutaveis) {
  lancesSelecionados.forEach(lanceSelecionado → {
      if (Objects.equals(lance.id(), lanceSelecionado.id())) {
          lancesSelecionados.remove(lanceSelecionado.id());
      }
  });
  if (lance.quantidade() ⩽ energiaRestante) {
      lancesSelecionados.add(lance);
      energiaRestante -= lance.quantidade();
  }
}
```

Escolhendo melhor candidato:

- Agora, para ambos os gulosos, percorremos os lances, fazemos uma pequena verificação antes, para ver se há algum lance duplicado e depois vamos adicionando lance por lance na solução até atingir o limite de energia restante. Já que a lista está ordenada para os melhores candidatos, entende-se que os melhores cadidatos serão sempre os primeiros.

```
int valorTotal = ZERO;
resultado.setLancesSelecionados(new ArrayList<>(lancesSelecionados));
```

Preenchendo a classe "MelhorResultado":

- Após a escolha dos candidatos para a solução preenchemos a classe `resultado` do tipo `MelhorResultado`, para que possa ser tirado as métricas de tempo de execução e resultados, preenchendo os logs.

```
for (Lance lance : lancesSelecionados) {
  valorTotal += lance.valor();
}
```

Calculando valor total arrecadado com os lances selecionados:

- Nessa parte vamos iterar por todos os lances que foram colocados na solução e somar o valor entre eles, para que tenhamos o valor total que conseguimos.

```
resultado.setLucroMaximizado(valorTotal);
resultado.setQuantidadeVendida(resultado.getProdutora().quantidadeDisponivel() - energiaRestante);
```

Preenchendo mais métricas da classe "resultado":

- Na última parte do código, vamos atualizar mais alguma métricas da classe `resultado` do tipo `MelhorResultado`, para agora ela ter os valores do lucro dos lances selecionados e da quantidade de megawatts vendidos.

Massa de testes utilizada

A massa de testes utilizada seguiu os seguintes parâmetros:

- **Quantidade *mínima* p/ compradora = 1000** → indica a quantidade mínima que uma determinada compradora poderia solicitar em um lote;
- **Quantidade *máxima* p/ compradora = 1500** → indica a quantidade máxima que uma determinada compradora poderia solicitar em um lote;
- **Quantidade disponível pela produtora = 8000** → indica a quantidade total (lote de megawatts) que a empresa produtora possui, ou seja, que disponibiliza para leilão;
- **Quantidade máxima de lances p/ compradora = 1** → indica a quantidade máxima de lances que cada compradora poderia fazer;
- **Quantidades de compradoras = [10, 11, ..., 33, 66, 99, ..., 330]** → foram executados 10 testes para cada quantidade de lances, iniciado com os valores usados no backingtrack, na prática foi do 10, até o 33. Após isso como pedido no enunciado, para o método guloso depois de usar o conjunto do backingtrack vamos ir icrementando de T em T até chegar em 10T, ou seja chegando até 330.

Os resultados gerados após cada execução do algoritmo foram armazenados automaticamente em dois arquivos: `exec-log.xls` e `hist-log.xls`.

Resultados obtidos

Quantidade lances	1	2	3	4	5	6	7	8	9	10	Tempo médio (seg)
10	-	-	-	-	-	-	-	-	-	-	-
11	-	-	-	-	-	-	-	-	-	-	-
12	-	-	-	-	-	-	-	-	-	-	-
13	-	-	-	-	-	-	-	-	-	-	-
14	-	-	-	-	-	-	-	-	-	-	-
15	-	-	-	-	-	-	-	-	-	-	-
16	-	-	-	-	-	-	-	-	-	-	-
17	-	-	-	-	-	-	-	-	-	-	-
18	-	-	-	-	-	-	-	-	-	-	-
19	-	-	-	-	-	-	-	-	-	-	-

Quantidade lances	1	2	3	4	5	6	7	8	9	10	Tempo médio (seg)
20	-	-	-	-	-	-	-	-	-	-	-
21	-	-	-	-	-	-	-	-	-	-	-
22	-	-	-	-	-	-	-	-	-	-	-
23	-	-	-	-	-	-	-	-	-	-	-
24	-	-	-	-	-	-	-	-	-	-	-
25	-	-	-	-	-	-	-	-	-	-	-
26	-	-	-	-	-	-	-	-	-	-	-
27	-	-	-	-	-	-	-	-	-	-	-
28	-	-	-	-	-	-	-	-	-	-	-
29	-	-	-	-	-	-	-	-	-	-	-
30	-	-	-	-	-	-	-	-	-	-	-
31	-	-	-	-	-	-	-	-	-	-	-
32	-	-	-	-	-	-	-	-	-	-	-
33	-	-	-	-	-	-	-	-	-	-	-

Quantidade lances	1	2	3	4	5	6	7	8	9	
10	11373	7979	9690	8233	8593	7382	10421	10152	10625	1
11	11034	10498	8851	9739	9525	8933	10969	11282	9600	1
12	9287	11649	11405	10123	11552	10650	11954	10264	11294	1
13	9694	10242	10502	11070	9776	10217	11494	9588	9550	9
14	11376	12223	13151	11436	11933	12961	10739	10762	12741	1
15	11980	11640	11667	10924	12055	13057	11062	11722	12633	1
16	10174	10711	12302	12198	13240	10984	11936	13242	10763	1
17	13256	12865	12268	11406	12471	13570	13926	14236	12900	1
18	12652	13160	13949	12861	14601	14589	13020	14125	13385	1
19	12471	13830	13950	11746	11199	11299	14364	14032	11408	1
20	13252	13461	15293	13320	13926	12878	13338	13779	14990	1

Quantidade lances	1	2	3	4	5	6	7	8	9	
21	15069	12409	13197	14481	12673	14401	12503	15206	12649	12
22	13404	13534	12706	11492	14486	14715	14426	13985	13569	13
23	13843	14227	13389	15164	13980	12742	14513	15229	14538	10
24	12990	15715	14265	15805	16449	13316	16848	14647	14109	13
25	13958	14033	13230	14286	14908	16462	12963	16322	15579	15
26	12647	14781	13987	15097	14160	15825	15016	14483	14062	12
27	15255	16195	15806	14696	16197	13267	14391	16503	15580	10
28	16552	13464	12779	16476	14725	16313	15567	13465	13332	15
29	16977	16617	13813	16096	15381	14058	16985	17193	13391	15
30	13862	15065	16791	13875	16640	15806	16700	15326	14675	17
31	15303	15520	13814	13830	16383	17212	16348	15900	14158	15
32	16207	14328	13321	14457	14690	15398	16212	15700	12567	15
33	14939	15697	16765	16564	16481	15791	16434	17319	14719	17
66	18069	19098	20009	17933	21272	19858	17655	19027	20518	20
99	18942	23146	20064	19294	19984	19330	20082	21291	23826	22
132	24657	23895	23221	20979	22289	20868	22302	21712	23965	20
165	24915	24435	22408	24039	24941	25050	21450	24634	20232	20
198	21950	24756	24934	23646	23989	22915	21796	25094	24603	22
231	24985	26065	26671	22663	26924	21840	21754	22187	26256	25
264	27361	28017	23116	25198	25295	24382	24939	24160	24478	27
297	23560	26667	23420	23139	21290	22079	24671	24880	23077	27
330	22185	28243	26234	25081	25389	23214	24349	28703	28072	23

Resultados obtidos Guloso2

Quantidade lances	1	2	3	4	5	6	7	8	9	10	Tempo médio (seg)
10	-	-	-	-	-	-	-	-	-	-	-
11	-	-	-	-	-	-	-	-	-	-	-

Quantidade lances	1	2	3	4	5	6	7	8	9	10	Tempo médio (seg)
12	-	-	-	-	-	-	-	-	-	-	-
13	-	-	-	-	-	-	-	-	-	-	-
14	-	-	-	-	-	-	-	-	-	-	-
15	-	-	-	-	-	-	-	-	-	-	-
16	-	-	-	-	-	-	-	-	-	-	-
17	-	-	-	-	-	-	-	-	-	-	-
18	-	-	-	-	-	-	-	-	-	-	-
19	-	-	-	-	-	-	-	-	-	-	-
20	-	-	-	-	-	-	-	-	-	-	-
21	-	-	-	-	-	-	-	-	-	-	-
22	-	-	-	-	-	-	-	-	-	-	-
23	-	-	-	-	-	-	-	-	-	-	-
24	-	-	-	-	-	-	-	-	-	-	-
25	-	-	-	-	-	-	-	-	-	-	-
26	-	-	-	-	-	-	-	-	-	-	-
27	-	-	-	-	-	-	-	-	-	-	-
28	-	-	-	-	-	-	-	-	-	-	-
29	-	-	-	-	-	-	-	-	-	-	-
30	-	-	-	-	-	-	-	-	-	-	-
31	-	-	-	-	-	-	-	-	-	-	-
32	-	-	-	-	-	-	-	-	-	-	-
33	-	-	-	-	-	-	-	-	-	-	-

Quantidade lances	1	2	3	4	5	6	7	8	9	
10	11373	7686	9690	8762	9025	7616	10421	10152	10625	1
11	11045	9075	12153	10521	8059	8923	10979	9996	9422	9
12	11395	12191	8099	10324	10754	9181	11782	11081	9353	1

Quantidade lances	1	2	3	4	5	6	7	8	9	
13	11059	8752	9575	11760	8387	9134	10967	11953	9610	8
14	8704	14002	10546	13192	11501	10319	10965	13356	11369	1
15	13159	7156	11932	13756	9679	13143	13287	10458	11735	9
16	10809	13099	10995	12129	11165	11561	10497	9984	12375	1
17	12485	11416	12902	11583	13970	13004	15253	11832	11300	1
18	14117	10332	14539	10381	13201	10398	15050	15806	12254	1
19	11327	13577	12025	11683	13110	11310	11330	12626	10682	1
20	13744	13016	13823	10777	11701	12725	14372	12391	13158	1
21	11691	14687	11976	12356	14227	14321	11895	12875	12234	1
22	13444	12580	12363	12873	11506	10820	13526	13900	16270	1
23	12598	13694	14542	12930	11755	14941	13185	13192	14313	1
24	13254	12372	14405	14660	14249	15548	14480	13241	12694	1
25	15730	12735	11297	11445	15022	14937	14891	15836	12737	1
26	13517	13400	15554	15708	11158	13307	14075	11386	13703	1
27	15719	15102	16821	13701	14189	12792	15119	14594	12034	1
28	15512	14432	14294	13815	12155	13181	12088	16273	13374	1
29	14407	15210	15748	15705	15007	13428	12728	15384	13411	1
30	17150	15671	12158	14380	14830	12550	15426	17516	11659	1
31	15196	12916	15097	12605	14523	12835	15003	17231	13506	1
32	15265	13652	15598	15208	12067	17032	12536	12854	13121	1
33	16421	15685	14078	12844	15467	16899	14958	17798	14047	1
66	18577	17469	19923	17424	19169	18839	15515	16134	19597	1
99	21112	21572	18664	21841	16107	20752	17981	20593	17947	1
132	23222	18656	17001	20458	18473	25019	19410	23016	22981	2
165	22422	23594	21360	21816	21905	21157	24698	20521	20895	1
198	23353	23407	23320	20903	20185	21243	22932	20957	23181	2

Quantidade lances	1	2	3	4	5	6	7	8	9	
231	20998	22551	21209	23103	25777	22971	21847	24440	22000	24
264	24338	26266	23149	22174	22807	24970	23829	23555	24243	22
297	20174	22838	22762	20622	21907	21679	23076	23513	24620	23
330	22443	25800	21933	23440	23287	23378	23746	22497	27116	24

O que concluir sobre as execuções? Quando analisamos os resultados das execuções dos dois algoritmos gulosos percebemos ambos tem um tempo de execução muito bom, na tabela parece que os dados não foram calculados e estão vazios, mas isso aconteceu porque para todos os conjuntos de dados testados o tempo de execução não passou de 1 segundo, a tabela não registra tempos em milésimos por isso apenas podemos concluir que foir bem rápido. Quanto aos resultados, notamos que o que opta pelo melhor valor por megawatt tem o melhor lucro, dessa forma ele pegou os lances mais valiosos e que compensam mais para formar a solução, no outro tipo de guloso, muitas boas soluções possíveis foram descartadas porque ele estava apenas pegando os de menor megawatt e assim tentando agrupar o maior número de lances, o problema disso é que você pode até conseguir um bom número de lances, mas o valor deles somados é menor, arrecando menos lucro na sua solução.

Em comparação com outros algoritmos, nenhum deles teve um tempo de execução tão bom quanto o Guloso, porém conseguiram achar soluções mais lucrativas do que o guloso. No primeiro algoritmo guloso, que prezava pelos lances de maior valor por megawatt, tivemos um resultado não muito descrepente do que o melhor possível, isso é uma vantagem de se escolher um bom critério guloso, agora o segundo que apenas tentava agrupar a maior quantidade de lance, teve um lucro bem divergente daqueles algoritmos que garantem o melhor resultado.

Algoritmo de divisão e conquista

Dados de execução

- **Responsável**: Breno Rosa Almeida
- **Matrícula**: 734290
- **JDK**: Java 17
- **Processador**: AMD Ryzen 7 3750H, 2.3 Ghz, 8 cores e 12 threads, 16mb de cache
- **GPU**: Nvidia GTX 1650
- **RAM**: 12GB, 2400Ghz
- **Sistema Operacional**: Windows 10
- **IDE**: IntelliJ Ultimate

Sobre o algoritmo

O algoritmo de `Divisão e Conquista` é uma estratégia clássica de resolução de problemas que envolve três passos principais: dividir o problema em subproblemas menores, resolver esses subproblemas de forma recursiva, e então combinar as soluções dos subproblemas para obter a solução final do problema original. Este método é

especialmente eficaz em problemas que podem ser divididos em partes menores e que possuem uma maneira clara de combinar as soluções das partes para formar uma solução global.

Algoritmo implementado

```
@Override
public AlgoritmosEnums algoritmo() {
    return AlgoritmosEnums.DIVISAO_CONQUISTA;
}
```

Classe `DivisaoConquista` : Implementa a interface `Algoritmo` , especificando que este algoritmo é do tipo "Divisão e Conquista" retornando a enumeração correspondente através do método `algoritmo()` .

```
@Override
public void executar(MelhorResultado resultado, List<Lance> todosLances, List<Lance> lancesSelecionadc
    int quantidadeEnergiaDisponivel = resultado.getProdutora().quantidadeDisponivel();

    List<Lance> melhorSelecao = resolverDivisaoEConquista(todosLances, quantidadeEnergiaDisponivel);

    resultado.setLancesSelecionados(melhorSelecao);
    resultado.setLucroMaximizado(melhorSelecao.stream().mapToInt(Lance::valor).sum());
}
```

Este método executa o algoritmo, recebendo a lista de todos os lances, a quantidade de energia disponível e outros parâmetros relevantes. Ele utiliza o método `resolverDivisaoEConquista` para determinar a melhor seleção de lances e atualiza o objeto `resultado` com os lances selecionados e o lucro maximizado.

```
private List<Lance> resolverDivisaoEConquista(List<Lance> lances, int quantidadeEnergiaDisponivel) {
    if (lances.isEmpty() || quantidadeEnergiaDisponivel ⩽ 0) {
        return new ArrayList<>();
    }

    if (lances.size() == 1) {
        Lance lance = lances.get(0);
        if (lance.quantidade() ⩽ quantidadeEnergiaDisponivel) {
            List<Lance> result = new ArrayList<>();
            result.add(lance);
            return result;
        } else {
            return new ArrayList<>();
        }
    }

    int meio = lances.size() / 2;
    List<Lance> esquerda = resolverDivisaoEConquista(new ArrayList<>(lances.subList(0, meio)), quantic
    List<Lance> direita = resolverDivisaoEConquista(new ArrayList<>(lances.subList(meio, lances.size()

    return combinarListas(direita, esquerda, quantidadeEnergiaDisponivel);
```

```
}
```

Este método divide a lista de lances em duas partes, resolve cada parte recursivamente e então combina as soluções obtidas. Se a lista de lances estiver vazia ou se a quantidade de energia disponível for zero, retorna uma lista vazia. Se houver apenas um lance, verifica se ele pode ser aceito e retorna uma lista com ele ou uma lista vazia.

```
private List<Lance> combinarListas(List<Lance> direita, List<Lance> esquerda, int quantidadeEnergiaDis
    List<Lance> lancesSelecionados = new ArrayList<>();
    List<Lance> todosLances = new ArrayList<>(direita);
    todosLances.addAll(esquerda);

    // Ordenar por valor/quantidade (eficiência) para maximizar o lucro
    todosLances.sort((l1, l2) -> {
        double efficiency1 = (double) l1.valor() / l1.quantidade();
        double efficiency2 = (double) l2.valor() / l2.quantidade();
        return Double.compare(efficiency2, efficiency1);
    });

    final int[] quantidadeTotalInserida = {0};

    for (Lance lance : todosLances) {
        if (quantidadeTotalInserida[0] + lance.quantidade() ⩽ quantidadeEnergiaDisponivel) {
            lancesSelecionados.add(lance);
            quantidadeTotalInserida[0] += lance.quantidade();
        }
    }

    // Tentar encontrar melhorias ao remover lances de menor eficiência e adicionar de maior eficiênci
    boolean melhoria = true;
    while (melhoria) {
        melhoria = false;
        List<Lance> possiveisAdicionar = todosLances.stream()
                .filter(l -> !lancesSelecionados.contains(l) && l.quantidade() + quantidadeTotalInseri
                .collect(Collectors.toList());

        for (Lance lanceRemover : new ArrayList<>(lancesSelecionados)) {
            for (Lance lanceAdicionar : possiveisAdicionar) {
                if (lanceAdicionar.valor() > lanceRemover.valor() &&
                        quantidadeTotalInserida[0] - lanceRemover.quantidade() + lanceAdicionar.quanti
                    lancesSelecionados.remove(lanceRemover);
                    lancesSelecionados.add(lanceAdicionar);
                    quantidadeTotalInserida[0] = quantidadeTotalInserida[0] - lanceRemover.quantidade(
                    melhoria = true;
                    break;
                }
            }
            if (melhoria) {
                break;
            }
        }
    }

    // Tentar adicionar qualquer lance restante que ainda caiba na capacidade
    List<Lance> lancesNaoSelecionados = todosLances.stream()
            .filter(l -> !lancesSelecionados.contains(l))
            .collect(Collectors.toList());
```

```
        for (Lance lance : lancesNaoSelecionados) {
            if (quantidadeTotalInserida[0] + lance.quantidade() ⩽ quantidadeEnergiaDisponivel) {
                lancesSelecionados.add(lance);
                quantidadeTotalInserida[0] += lance.quantidade();
            }
        }

        return lancesSelecionados;
    }
```

Este método combina os resultados das duas listas (esquerda e direita) em uma solução única. Ele começa ordenando todos os lances pela eficiência (valor/quantidade) em ordem decrescente. Em seguida, tenta selecionar os lances mais eficientes sem exceder a quantidade de energia disponível. Depois, realiza uma melhoria iterativa, tentando substituir lances menos eficientes por lances mais eficientes até que não haja mais melhorias possíveis. Finalmente, tenta adicionar qualquer lance restante que ainda caiba na capacidade de energia disponível.

Massa de testes utilizada

Neste caso, utilize os mesmos conjuntos de tamanho T utilizados no backtracking.

Resultados obtidos

Qtde lances	1	2	3	4	5	6	7	8	9	10
10	11373	8394	9690	9040	10005	8368	10421	10152	10625	11697
11	11045	9075	12153	10521	8854	9458	11361	10629	10105	10587
12	11395	12706	9367	10763	11212	9854	12350	11432	9855	11932
13	11059	9279	9575	11873	9421	9821	11747	12300	10670	10045
14	9281	14021	10814	14062	11814	11063	11309	13356	12220	11493
15	13535	8200	13383	13756	11483	14188	13611	11064	12690	10956
16	12160	13591	12249	13478	11905	12510	12011	12417	12924	11580
17	13424	12573	14009	12169	14769	13004	16050	12196	12516	11776
18	14208	11297	15407	11440	14408	10804	15238	16301	13775	13227
19	12453	14352	13737	13724	13662	12215	12236	14697	11784	13330
20	14831	14268	15035	11828	13712	13552	14871	13827	13986	15356
21	11951	16218	13216	13626	15327	14926	13176	14002	13501	13957
22	14270	13253	13694	15215	13076	11853	14362	14574	16733	12014

Qtde lances	1	2	3	4	5	6	7	8	9	10
23	14160	14795	15002	13760	13577	16308	14030	14619	15402	14902
24	14003	13505	14812	15435	14425	15916	15243	14105	15037	14274
25	15837	14717	13223	13437	15213	15876	16253	17029	14090	14621
26	15217	14726	16423	17017	13700	14073	15667	12722	15887	14369
27	16934	15958	17389	14517	14770	14705	16279	15783	13216	16202
28	16909	15472	15263	14260	13573	14049	13686	17431	15361	15299
29	15956	16563	16741	17101	16104	14728	14132	17076	15299	15806
30	17738	17273	14179	16732	16281	14321	16194	18770	14250	15564
31	16662	14927	16779	14237	15641	14458	17013	18222	14349	16191
32	17290	15307	17305	15826	13945	17748	14959	14481	14309	14393
33	17028	16308	15647	14528	16205	17425	17212	18868	15911	15793
25	26725	40348								

[Tabela de maior lucro obtido - Divisão e Conquista]

Além dos cenários apresentados, foram executados dois conjuntos adicionais fornecidos pelo prof. Caram, conjunto um e dois. A relação das informações obtidas com a execução desses conjuntos são elencadas na tabela abaixo.

Conjunto	Quantidade lances	Lances selecionados	Lucro máximo	Tempo execução (seg)
Um	25	19	R$ 26.725,00	10
Dois	25	21	R$ 40.348,00	10

Algoritmo por programação dinâmica

Dados de execução

- **Responsável**: Marcos Paulo Freitas Da Silva
- **Matrícula**: 746639
- **JDK**: Java 17
- **Processador**: Intel Core i5 8265U, 4.2 Ghz, 4 cores e 8 threads, 6mb de cache
- **RAM**: 12GB, 2400Ghz
- **Sistema Operacional**: Windows 11
- **IDE**: IntelliJ Ultimate

Sobre o algoritmo

A classe `ProgramacaoDinamica` implementa o algoritmo de Programação Dinâmica, uma técnica de otimização que resolve problemas complexos dividindo-os em subproblemas menores e resolvendo cada subproblema apenas uma vez, armazenando seus resultados para evitar cálculos repetidos.

Para que o método da programação dinâmica possa ser aplicado, é preciso que o problema tenha estrutura recursiva, a solução de toda instância do problema deve conter soluções de subinstâncias da instância. A característica distintiva da programação dinâmica é a tabela que armazena as soluções das várias subinstâncias. Em alguns casos, o algoritmo recursivo refaz a solução de cada subinstância muitas vezes, e isso torna o algoritmo ineficiente. Nesses casos, é possível armazenar as solução das subinstância numa tabela e assim evitar que elas sejam recalculadas

Algoritmo implementado

```
@Override
public void executar(
        @NonNull MelhorResultado melhorResultado, @NotNull List<Lance> todosLances, @NonNull List<Lanc

    int quantidadeDisponivel = melhorResultado.getProdutora().quantidadeDisponivel();
    int n = todosLances.size();
    int capacidade = 8000;

    int[] dp = new int[capacidade + 1];
    int[] selecionados = new int[capacidade + 1];

    for (int i = 0; i < n; i++) {
        Lance lance = todosLances.get(i);
        int quantidade = lance.quantidade();
        int valor = lance.valor();

        for (int j = capacidade; j >= quantidade; j--) {
            if (dp[j - quantidade] + valor > dp[j]) {
                dp[j] = dp[j - quantidade] + valor;
                selecionados[j] = i;
            }
        }
    }

    int maxLucro = 0;
    int melhorCapacidade = 0;

    for (int i = 0; i <= capacidade; i++) {
        if (dp[i] > maxLucro) {
            maxLucro = dp[i];
            melhorCapacidade = i;
        }
    }

    List<Lance> melhoresLances = new ArrayList<>();
    int capacidadeAtual = melhorCapacidade;

    while (capacidadeAtual > 0 && selecionados[capacidadeAtual] != 0) {
        Lance lance = todosLances.get(selecionados[capacidadeAtual]);
        melhoresLances.add(lance);
        capacidadeAtual -= lance.quantidade();
```

```
        }

        melhorResultado.setLucroMaximizado(maxLucro);
        melhorResultado.setLancesSelecionados(melhoresLances);
    }
```

No algoritmo implementado, a função responsável por executar o método de ***Programção Dinamica*** recebe cinco parâmetros, sendo eles:

- melhorResultado: `MelhorResultado` ;
- todosLances: `List<Lance>` ;
- lancesSelecionados: `List<Lance>` ;
- indice: `int` ;
- lucroAtual: `int` .

`quantidadeDisponivel` armazena a capacidade total de venda. `n` representa o número total de lances. `capacidade` define a capacidade máxima (neste caso, 8000). `dp` é um array para armazenar os lucros máximos possíveis para cada capacidade. `selecionados` é um array para rastrear os lances que foram selecionados para formar o lucro máximo.

Construção da Tabela de Programação Dinâmica:

Inicializa variáveis para a quantidade disponível, o número de lances, e a capacidade máxima (8000). Cria dois arrays: dp para armazenar os valores máximos de lucro para cada capacidade, e selecionados para rastrear os índices dos lances selecionados.

```
int[] dp = new int[capacidade + 1];
int[] selecionados = new int[capacidade + 1];
```

Loop de Programação Dinâmica:

Esta parte do código itera sobre todos os lances e atualiza a tabela dp e o array selecionados. A tabela dp é preenchida de forma que dp[j] representa o lucro máximo que pode ser obtido com a capacidade j. Se incluir o lance atual i resulta em um lucro maior, a tabela é atualizada.

```
for (int i = 0; i < n; i++) {
  Lance lance = todosLances.get(i);
  int quantidade = lance.quantidade();
  int valor = lance.valor();

  for (int j = capacidade; j ⩾ quantidade; j--) {
    if (dp[j - quantidade] + valor > dp[j]) {
      dp[j] = dp[j - quantidade] + valor;
      selecionados[j] = i;
    }
  }
}
```

Determinação do Lucro Máximo:

O algoritmo itera sobre o array dp para encontrar o lucro máximo possível (`maxLucro`) e a capacidade correspondente (`melhorCapacidade`).

```
int maxLucro = 0;
int melhorCapacidade = 0;

for (int i = 0; i ⩽ capacidade; i++) {
  if (dp[i] > maxLucro) {
    maxLucro = dp[i];
    melhorCapacidade = i;
  }
}
```

Recuperação dos Lances Selecionados

Reconstrói a lista dos melhores lances utilizando o array selecionados. Começa na capacidade máxima e retrocede até encontrar todos os lances selecionados.

```
List<Lance> melhoresLances = new ArrayList<>();
int capacidadeAtual = melhorCapacidade;

while (capacidadeAtual > 0 && selecionados[capacidadeAtual] =/= 0) {
  Lance lance = todosLances.get(selecionados[capacidadeAtual]);
  melhoresLances.add(lance);
  capacidadeAtual -= lance.quantidade();
}
```

Atualização do Melhor Resultado:

O lucro máximo e a lista de lances selecionados são armazenados no objeto `melhorResultado` .

```
melhorResultado.setLucroMaximizado(maxLucro);
melhorResultado.setLancesSelecionados(melhoresLances);
```

O algoritmo de programação dinâmica implementado busca encontrar a combinação de lances que maximiza o lucro total, respeitando a capacidade de venda da empresa produtora. O algoritmo realiza as seguintes etapas:

- Inicializa variáveis e arrays de controle;
- Constrói uma tabela de programação dinâmica para determinar os lucros máximos possíveis para cada capacidade;
- Identifica o lucro máximo e a capacidade correspondente;
- Rastreia os lances que compõem o lucro máximo;
- Atualiza o objeto melhorResultado com o lucro máximo e a lista de lances selecionados;
- Ao final da execução, o algoritmo retorna a melhor combinação de lances encontrada, maximizando o lucro total.

Massa de testes utilizada

Aqui utilizaremos a mesma massa de testes presente no algoritmo guloso.

Resultados obtidos

Conforme descrito acima, os cenários de testes variaram de 10 a 31 lances, para cada um desses cenários, foram executados 10 iterações com conjuntos distintos de lances, gerados em tempo de execução, de modo que foram executadas * *210** execuções ao total (ou seja, (31 - 10) * 10). Nos primeiros testes realizados com o conjunto de lances gerados para o backtracking, o tempo de carregamento dos lances foi de 0,00 segundos, ao passo que o valor do melhor resultado continua o mesmo encontrado no backtracking. A tabela a seguir apresenta os resultados obtidos com a execução do algoritmo de programação dinâmica.

Quantidade lances	1	2	3	4	5	6	7	8	9
10	11.373	8.394	9.690	9.040	10.005	8.368	10.421	10.152	10.625
11	11.045	9.075	12.153	10.521	8.854	9.458	11.361	10.629	10.105
12	11.395	12.706	9.367	10.763	11.212	9.854	12.350	11.432	9.855
13	11.059	9.279	9.575	11.873	9.421	9.821	11.747	12.300	10.670
14	9.281	14.021	10.814	14.062	11.814	11.063	11.309	13.356	12.220
15	13.535	8.200	13.383	13.756	11.483	14.188	13.611	11.064	12.690
16	12.160	13.591	12.249	13.478	11.905	12.510	12.011	12.417	12.924
17	13.424	12.573	14.009	12.169	14.769	13.004	16.050	12.196	12.516
18	14.208	11.297	15.407	11.440	14.408	10.804	15.238	16.301	13.775
19	12.453	14.352	13.737	13.724	13.662	12.215	12.236	14.697	11.784
20	14.831	14.268	15.035	11.828	13.712	13.552	14.871	13.827	13.986
21	11.951	16.218	13.216	13.626	15.327	14.926	13.176	14.002	13.501
22	14.270	13.253	13.694	15.215	13.076	11.853	14.362	14.574	16.733
23	14.160	14.795	15.002	13.760	13.577	16.308	14.030	14.619	15.402
24	14.003	13.505	14.812	15.435	14.425	15.916	15.243	14.105	15.037
25	15.837	14.717	13.223	13.437	15.213	15.876	16.253	17.029	14.090
26	15.217	14.726	16.423	17.017	13.700	14.073	15.667	12.722	15.887
27	16.934	15.958	17.389	14.517	14.770	14.705	16.279	15.783	13.216
28	16.909	15.472	15.263	14.260	13.573	14.049	13.686	17.431	15.361

Quantidade lances	1	2	3	4	5	6	7	8	9
29	15.956	16.563	16.741	17.101	16.104	14.728	14.132	17.076	15.299
30	17.738	17.273	14.179	16.732	16.281	14.321	16.194	18.770	14.250
31	16.662	14.927	16.779	14.237	15.641	14.458	17.013	18.222	14.349
32	17.290	15.307	17.305	15.826	13.945	17.748	14.959	14.481	14.309
33	17.028	16.308	15.647	14.528	16.205	17.425	17.212	18.868	15.911

[Tabela de maior lucro obtido 1 - Programação Dinamica]

Quantidade lances	1	2	3	4	5	6	7	8	9
66	20.445	23.373	25.110	24.980	26.100	25.631	28.143	25.192	28.326
99	19.769	24.297	21.451	26.267	25.247	27.102	29.552	26.622	29.565
132	22.405	20.979	20.596	25.494	25.807	24.197	28.104	27.217	26.298
165	19.741	23.520	23.200	25.607	25.225	26.436	27.455	25.890	27.733
198	20.925	18.671	21.397	26.468	23.764	28.959	26.325	27.401	28.608
231	21.437	22.761	27.172	24.485	27.107	27.331	27.871	27.260	28.751
264	18.557	19.753	22.452	26.392	26.240	26.475	28.379	27.560	29.330
297	19.056	22.694	25.891	24.703	25.659	28.645	26.363	27.636	26.854
330	20.432	20.185	24.409	23.142	26.413	25.418	28.133	29.693	28.512

[Tabela de maior lucro obtido 2 - Programação Dinamica]

Além dos cenários apresentados, foram executados dois conjuntos adicionais fornecidos pelo prof. Caram, conjunto um e dois. A relação das informações obtidas com a execução desses conjuntos são elencadas na tabela abaixo.

Conjunto	Quantidade lances	Lances selecionados	Lucro máximo	Tempo execução (seg)
Um	25	19	R$ 26.725,00	0
Dois	25	16	R$ 40.348,00	0

Conclusão

Este algoritmo implementa uma solução clássica de programação dinâmica para maximizar o lucro dentro de uma capacidade limitada (similar ao problema da mochila). Ele itera sobre os lances disponíveis e preenche uma tabela (`dp`) para rastrear o lucro máximo possível para cada capacidade, e um array (`selecionados`) para rastrear quais lances foram escolhidos para alcançar esse lucro. Ao final, ele reconstrói a lista de lances selecionados e atualiza o objeto `melhorResultado` com o lucro máximo e os lances correspondentes.

Comparação dos resultados obtidos pelos algoritmos

A seguir serão apresentados comparativos entre os quatro algoritmos implementos sob óticas distintas, como resultado obtido, tempo de execução, dentre outros.

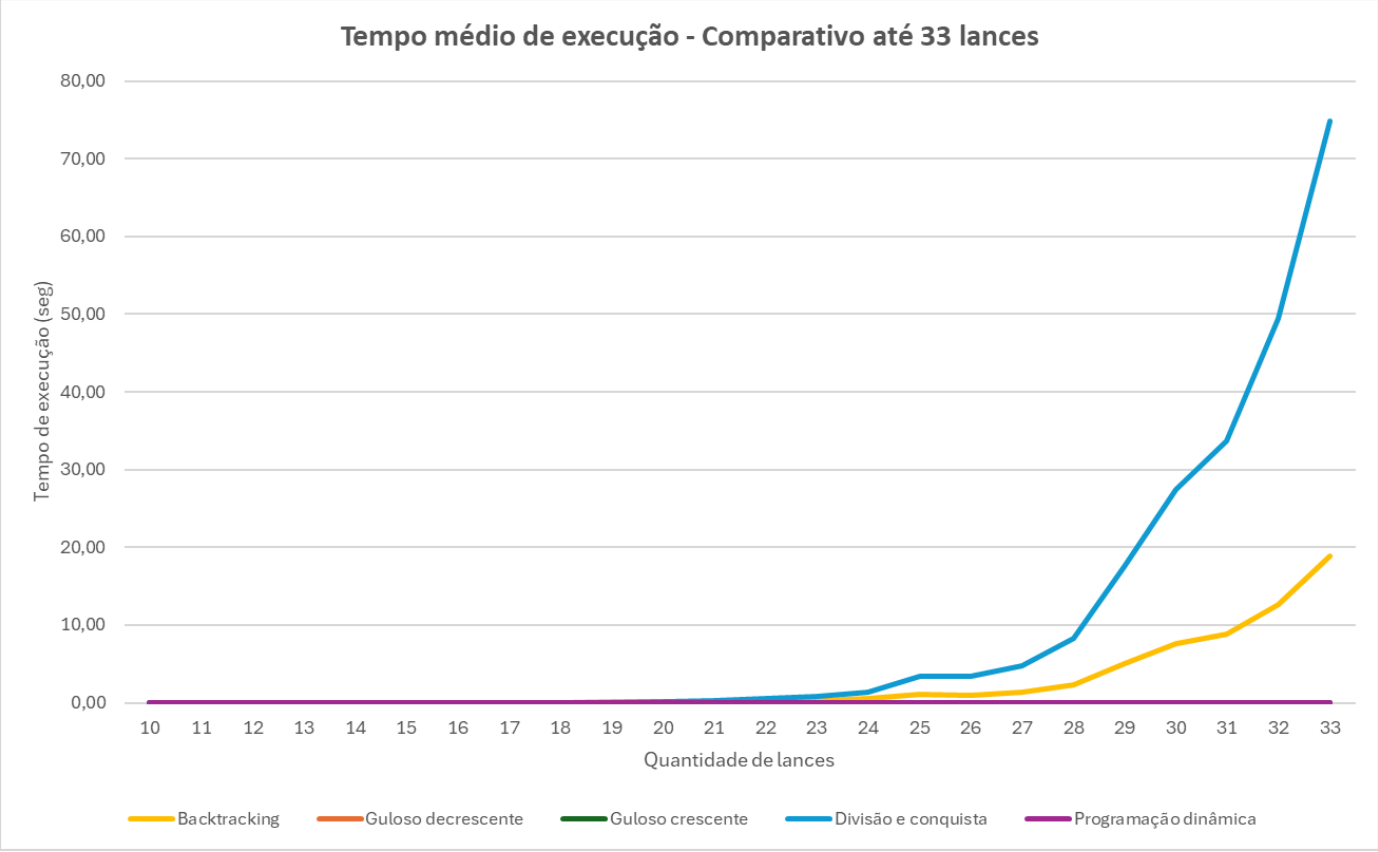
Tempo de execução

A análise do tempo será feita em duas etapas, primeiro comparando o trecho comum de execução entre todos os algoritmos, que vai dos casos de testes com 10 a 33 lances, e em seguida comparando os algoritmos que foram até 330 lances (Guloso e Programação dinâmica). Para a primeira etapa (10 a 33 lances), o algoritmo de Divisão e Conquista foi a implementação que apresentou o pior desempenho, com tempo médio por execução de 9,42 segundos, seguido pela implementação do Backtracking, com tempo médio por execução de 2,5 segundos. Para esse cenário, as implementações do algoritmo Guloso e Programação dinâmica apresentaram tempo de execução inferior a 1 segundo, estando indicado pelo símbolo "-" na tabela abaixo.

Qtde lances	Backtracking	Guloso decrescente	Guloso crescente	Divisão e conquista	Programação dinâmica
10	-	-	-	-	-
11	-	-	-	-	-
12	-	-	-	-	-
13	-	-	-	-	-
14	-	-	-	-	-
15	-	-	-	-	-
16	-	-	-	-	-
17	-	-	-	-	-
18	-	-	-	-	-
19	0,10	-	-	-	-
20	0,10	-	-	0,10	-
21	0,10	-	-	0,30	-
22	0,10	-	-	0,50	-

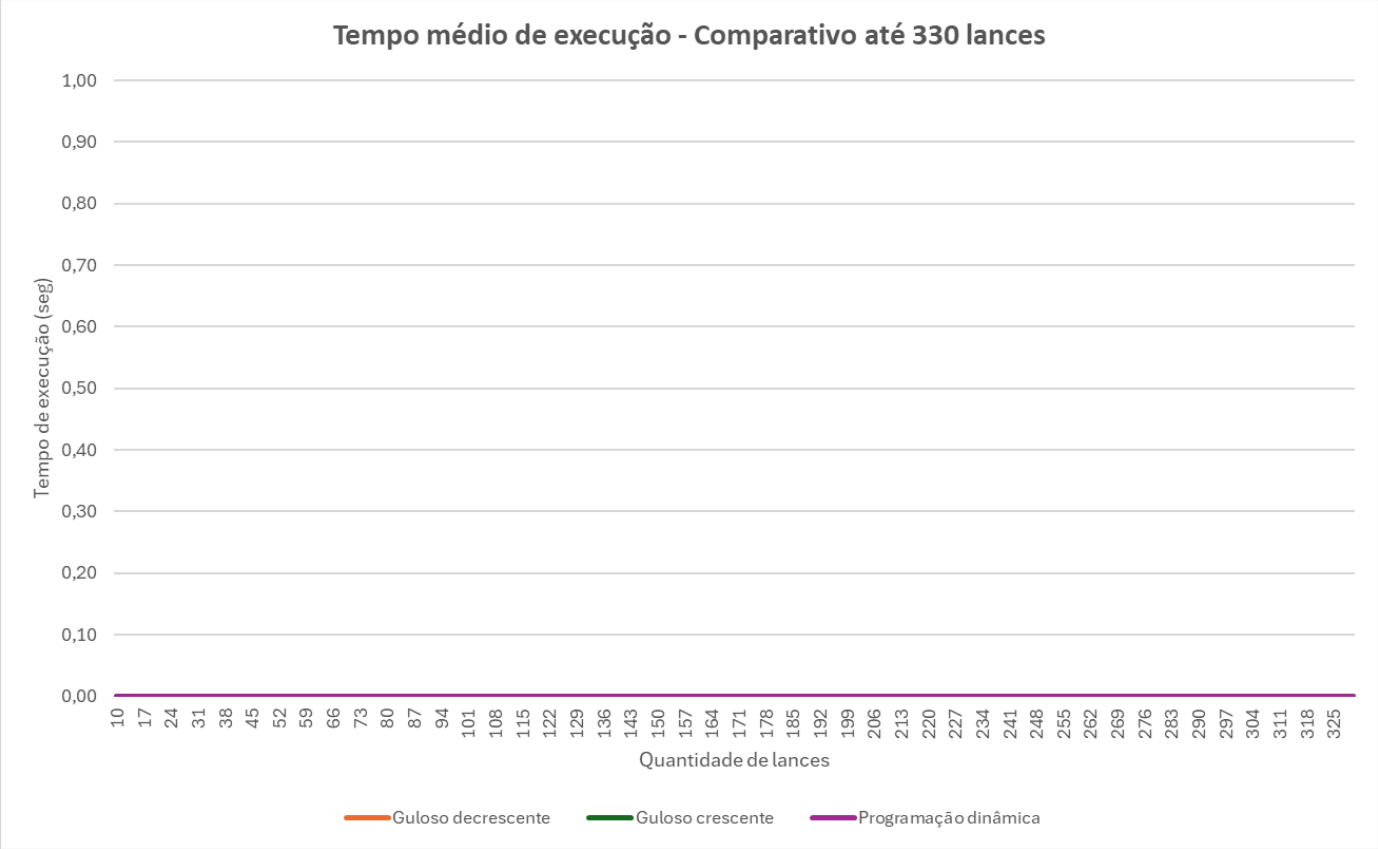
Qtde lances	Backtracking	Guloso decrescente	Guloso crescente	Divisão e conquista	Programação dinâmica
23	0,30	-	-	0,80	-
24	0,50	-	-	1,40	-
25	1,08	-	-	3,42	-
26	1,00	-	-	3,40	-
27	1,40	-	-	4,80	-
28	2,30	-	-	8,30	-
29	5,00	-	-	17,50	-
30	7,60	-	-	27,50	-
31	8,80	-	-	33,70	-
32	12,70	-	-	49,40	-
33	18,90	-	-	74,90	-
Tempo médio	2,50	-	-	9,42	-

[Tabela de comparação de desempenho de 10 a 33 lances]



[Gráfico de comparação de desempenho de 10 a 33 lances]

Para a segunda etapa, comparamos os algoritmos que executaram até o cenário com 330 lances (Guloso e Programação dinâmica). Para ambas implementações o tempo de execução foi inferior a 1 segundo, conforme pode ser verificado no gráfico a seguir.



[Gráfico de comparação de desempenho de 10 a 330 lances]

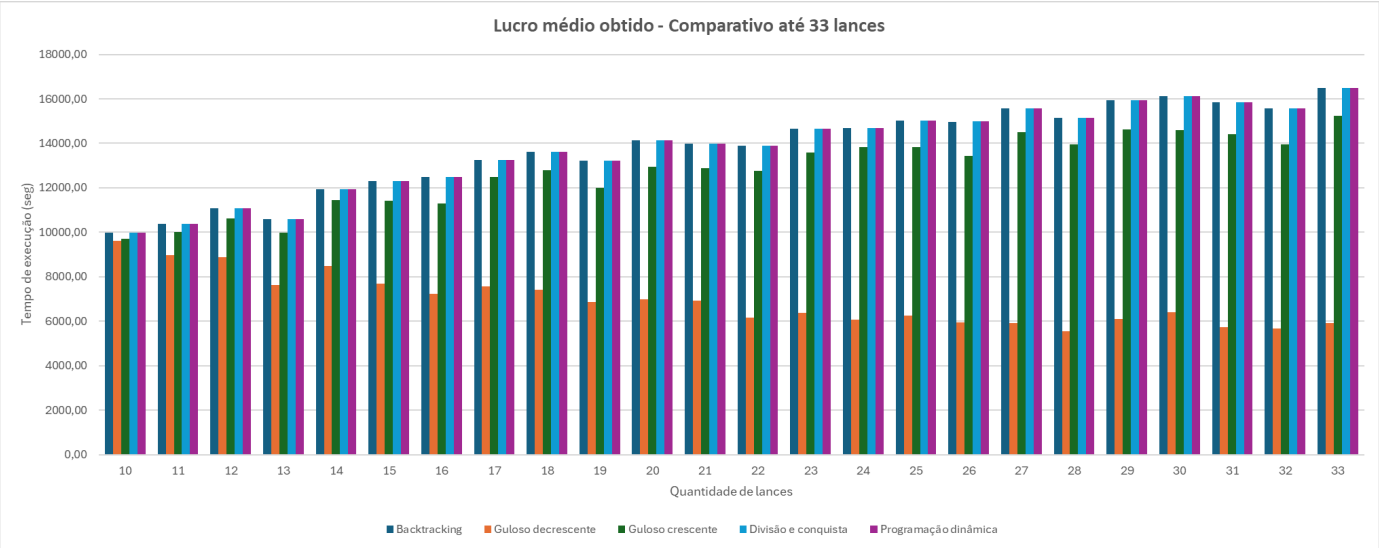
Lucro encontrado

De maneira análoga à análise do tópico acima, a análise do lucro encontrado também será dividida em duas etapas, primeiro comparando o trecho comum de execução entre todos os algoritmos, que vai dos casos de testes com 10 a 33 lances, e em seguida comparando os algoritmos que foram até 330 lances (Guloso e Programação dinâmica). Para a primeira etapa, conseguimos verificar que o lucro encontrado pelos algoritmos de Backtracking, Divisão e conquista e Programação se manteve o mesmo durante todos os casos de teste, o que se é esperado para estes algoritmos. A implementação do algoritmo Guloso que utilizou como metodologia a ordenação dos lances por ordem crescente apresentou resultado satisfatório, haja visto que na média apresentou um lucro máximo 7% abaixo do resultado ótimo (lucro máximo possível) e possui um tempo de execução muito baixo. A implementação do algoritmo Guloso que ordenou os lances por ordem decrescente não apresentou resultado satisfatório, com lucro médio 50% inferior ao resultado ótimo esperado.

Qtde lances	Backtracking	Guloso decrescente	Guloso crescente	Divisão e conquista	Programação dinâmica
10	9.976,50	9.614,50	9.704,70	9.976,50	9.976,50

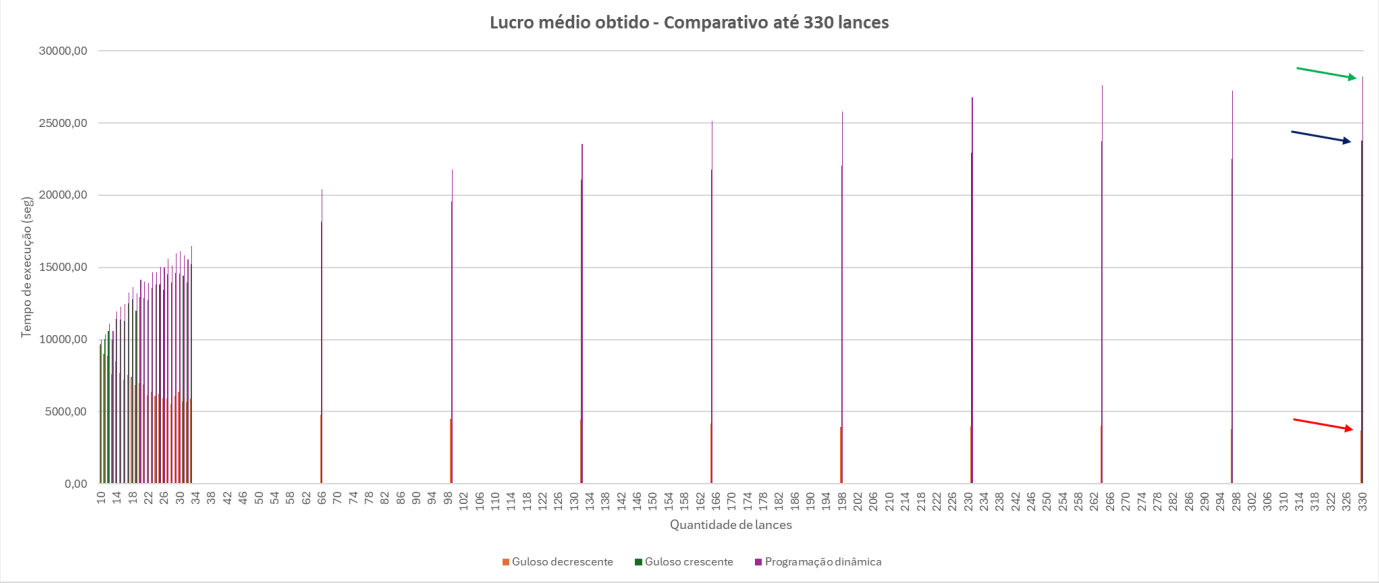
Qtde lances	Backtracking	Guloso decrescente	Guloso crescente	Divisão e conquista	Programação dinâmica
11	10.378,80	8.977,90	10.011,60	10.378,80	10.378,80
12	11.086,60	8.877,10	10.609,20	11.086,60	11.086,60
13	10.579,00	7.612,30	9.981,00	10.579,00	10.579,00
14	11.942,60	8.488,10	11.448,50	11.943,30	11.943,30
15	12.286,60	7.688,50	11.405,90	12.286,60	12.286,60
16	12.482,50	7.240,80	11.278,40	12.482,50	12.482,50
17	13.248,60	7.553,20	12.493,00	13.248,60	13.248,60
18	13.610,50	7.416,50	12.802,30	13.610,50	13.610,50
19	13.219,00	6.850,30	12.006,60	13.219,00	13.219,00
20	14.126,60	6.988,60	12.928,80	14.126,60	14.126,60
21	13.990,00	6.908,70	12.886,20	13.990,00	13.990,00
22	13.904,40	6.159,30	12.760,80	13.904,40	13.904,40
23	14.655,50	6.360,20	13.578,40	14.655,50	14.655,50
24	14.675,50	6.081,20	13.841,30	14.675,50	14.675,50
25	15.029,60	6.235,50	13.836,20	15.029,60	15.029,60
26	14.966,40	5.937,80	13.428,70	14.980,10	14.980,10
27	15.575,30	5.917,50	14.507,60	15.575,30	15.575,30
28	15.130,30	5.540,50	13.950,20	15.130,30	15.130,30
29	15.950,60	6.108,20	14.618,90	15.950,60	15.950,60
30	16.130,20	6.399,30	14.582,40	16.130,20	16.130,20
31	15.847,90	5.727,70	14.422,40	15.847,90	15.847,90
32	15.556,30	5.678,30	13.943,60	15.556,30	15.556,30
33	16.492,50	5.919,80	15.233,80	16.492,50	16.492,50
Lucro médio	13.785,08	6.928,41	12.760,85	13.785,68	13.785,68
% lucro máximo	100%	50%	93%	100%	100%

[Tabela de comparação de valores de 10 a 33 lances]



[Gráfico de comparação de valores de 10 a 33 lances]

Comparando os algoritmos que seguiram a execução até 330 lances, podemos observar que o desempenho de ambas as implementações do algoritmo guloso caíram. Para a implementação por ordem crescente (seta azul no gráfico), o lucro na última execução apresentou 84% do lucro máximo obtido pela Programação dinâmica (seta verde no gráfico), enquanto a implementação por ordem decrescente apresentou apenas 13% do lucro máximo obtido pela Programação dinâmica (seta vermelha no gráfico).



[Gráfico de comparação de valores de 10 a 330 lances]

Conjuntos solicitados pelo prof. Caram

Conforma informado anteriormente, foram disponibilizados dois conjuntos adicionais pelo professor para que fossem realizados testes com eles. Cada um desses conjuntos possuía 25 lances, com combinações diferentes de quantidade e valor por lance. Para fins de análise, esses conjuntos serão referenciados como conjuntos 1 e 2. Conforme pode ser observado na tabela a seguir, o lucro máximo obtido pelos algoritmos de Backtracking, Guloso crescente, Divisão e conquista, e Programação dinâmica para o conjunto 1 foi de R$ 26.725,00. O Guloso decrescente, por sua vez, apresentou lucro final de R$ 24.219,00.

Os algoritmos de Backtracking e Divisão e conquista apresentaram os maios tempos de execução para esses cenários, despendendo 4 e 10 segundos para o conjunto 1, e 3 e 10 segundos para o conjunto 2, respectivamente.

Algoritmo	Nº Conjunto	Qtde lances	Tempo (seg)	Lances selecionados	Lucro obtido
Backtracking	1	25	4	19	26.725,00
Guloso decrescente	"	"	0	19	24.219,00
Guloso crescente	"	"	0	19	26.725,00
Divisão e conquista	"	"	10	19	26.725,00
Programação dinâmica	"	"	0	19	26.725,00
Backtracking	2	25	3	21	40.348,00
Guloso decrescente	"	"	0	21	36.916,00
Guloso crescente	"	"	0	22	39.408,00
Divisão e conquista	"	"	10	21	40.348,00
Programação dinâmica	"	"	0	16	40.348,00

[Tabela de comparação de valores e tempos dos conjuntos 1 e 2]

Comentários finais sobre os resultados encontrados

Com base nos resultados obtidos, foi possível verificar que, de modo geral, em conjuntos pequenos, a escolhe de um algoritmo mais ou menos eficaz não se mostra muito expressiva, haja vista que os tempos de execução tendem a ser inferiores a 1 segundo. Nos nossos casos de testes, pode-se dizer que em um conjunto de aproximadamente 20 a 23 dados, o algoritmo implementado não é muito significante. No entanto, a partir desse ponto, a diferença de desempenho começou a ser representativa, ao passo que com mais de 33 lances, por exemplo, tínhamos algoritmos gastando mais de 1 minuto para executar o conjunto, enquanto outros executavam até 330 lances em menos de 1 segundo.