UNIVERSIDADE ESTADUAL DE MARINGÁ

NÍCOLAS DOS SANTOS CARVALHO

TRABALHO DE ALGORÍTIMOS GULOSOS E PROGRAMAÇÃO DINÂMICA

MARINGÁ

2023

NÍCOLAS DOS SANTOS CARVALHO

TRABALHO DE ALGORÍTIMOS GULOSOS E PROGRAMAÇÃO DINÂMICA

Trabalho de Algorítimos Gulosos e Programação Dinâmica apresentado à disciplina de Projeto e Análise de Algoritmos, do Bacharelado em Ciência da Computação do Departamento de Informática

Docente: Prof. Dr. Daniel Kikuti

MARINGÁ
2023

1 1970 - PRIMEIRO CONTATO

O problema de número mil novecentos e setenta do *beecrowd*, que pode ser acessado em: https://www.beecrowd.com.br/judge/pt/problems/view/1970. Propõe que dada três linhas de entradas contendo, em ordem: o número de músicas e fitas; a duração das músicas; a capacidade das fitas em minutos. Deve ser feito um algorítimo que retorne qual o máximo de minutos que podem ser gravados nas fitas.

Exemplo

Entradas

8 3

7 3 3 2 4 4 2 3

9 8 9

Saída

26

1.1 Formulação Recursiva

Primeiramente para entender como se pode formular recursivamente o problema é necessário entender as possibilidades de cada iteração do algorítimo. Para cada análise em um determinado estado pode-se escolher ou não colocar uma música em um dos cartucho, ou seja descartar, ou incluir a música na solução do problema, e assim deve-se fazer até que não sobrem mais músicas.

Sendo

- n = numero de musicas - 1;
- k = numero de cartuchos;
- mus = arranjo de músicas de $[0..n]$;
- $cart$ = arranjo dos cartuchos de $[0..k-1]$;
- $novos\_cart_i$ = uma cópia de $cart$ com $cart[i]$ -= $mus[n]$

A formulação pode ser interpretada como:

$$
f(mus, cart, n, k) = \begin{cases}
0 & \text{se } n < 0 \\
f(mus, cart, n-1, k) & \text{se } k = 1 \; e \; mus[n] > cart[0] \\
 & \text{ou } \; k = 2 \; e \\
 & \quad (mus[n] > cart[0] \\
 & \quad e \; mus[n] > cart[1]) \\
 & \text{ou } \; k = 3 \; e \\
 & \quad (mus[n] > cart[0] \\
 & \quad e \; mus[n] > cart[1] \\
 & \quad e \; mus[n] > cart[2]) \\
max \; \{ & \text{se } k = 1 \\
\quad f(mus, cart, n-1, k), & \text{e } mus[n] <= cart[0] \\
\quad f(mus, novos\_cart_0, n-1, k) & \text{[*]} \\
\} & \\
max \; \{ & \text{se } k = 2 \\
\quad f(mus, cart, n-1, k), & \text{e } mus[n] <= cart[0] \\
\quad f(mus, novos\_cart_0, n-1, k), & \text{e } mus[n] <= cart[1] \\
\quad f(mus, novos\_cart_1, n-1, k) & \text{[*]} \\
\} & \\
max \; \{ & \text{se } k = 3 \\
\quad f(mus, cart, n-1, k), & \text{e } mus[n] <= cart[0] \\
\quad f(mus, novos\_cart_0, n-1, k), & \text{e } mus[n] <= cart[1] \\
\quad f(mus, novos\_cart_1, n-1, k), & \text{e } mus[n] <= cart[2] \\
\quad f(mus, novos\_cart_2, n-1, k) & \text{[*]} \\
\} &
\end{cases}
$$

[*]: *por fins de legibilidade foi ocultado as chamadas como* $se\ k = 3$ *e algum cartucho é menor que a música, mas nesses casos a chamada recursiva para esse cartucho deve ser ignorada.*

Note que quando $n >= 0\ e\ k > 0$, todas as possibilidades são testadas quanto a adicionar a música no cartucho 0, 1, 2 e é retornado sempre a maior solução delas, que indica o máximo de minutos que dá para inserir com aquela música no conjunto solução. Não obstante disso também é testado para caso a música seja ignorado em $f(mus, cart, n-1, k)$, pois não há alteração da lista dos cartuchos, assim como não é modificado a quantidade de cartuchos, apenas é testado para o próximo. Nota: as soluções que envolvem um cartucho que não há espaço para a $mus[n]$ não são chamadas recursivamente, pois se sobrepõem com a solução de caso a música seja ignorada.

1.1.1 Exemplo e Subestrutura Ótima

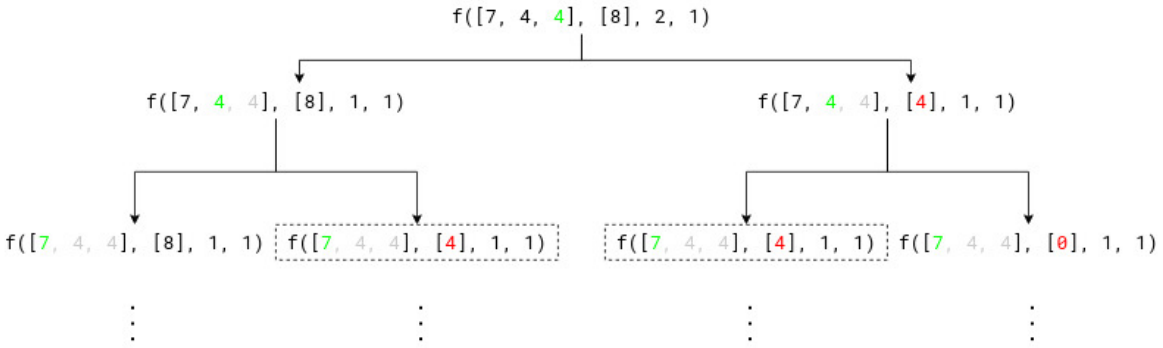
Por exemplo ao escolher as entradas $n = 2, k = 1$ e $mus = [6, 4, 3]$; $cart = [8]$ a primeira chamada que é feita será $f(mus, cart, n - 1, k)$ como n é maior ou igual a 0 e k é maior que 0, se chamado para o próximo sem incluir $mus[2] = 3$ na função $f(mus, cart, (n - 1) - 1, k)$ o retorno será o caso ótimo sem a inclusão de $mus[2]$. No entanto além da chamada descartando, há chamadas incluindo em um dos cartuchos que cabe 3 como em $cart[0]$ que se 3 for deduzido o novo conjunto de cartuchos, ou $novos\_cart_0$ será $[5]$ e a chamada recursiva $f(mus, novos\_cart_0, (n-1)-1, k)$ retornará a solução ótima incluindo $mus[2]$. a função max verificará e retornará qual dos 2 casos retorna o maior valor, assim recursivamente até o caso base. Por esse caráter de sempre escolher as melhores opções a partir da base, se comprova a subestrutura ótima do problema.

Vale ressaltar, que o caso base ocorre quando $n < 0$, que indica que todas as músicas já foram incluídas na solução, seja ela em um dos cartuchos ou descartadas.

1.1.2 Sobreposição de problemas

Sobre a sobreposição de problemas é possível notar que ela ocorre principalmente pelo caráter recursivo de múltiplas chamadas que acontecem. Como se pode notar no gráfico a seguir as chamadas $f([7,4,4],[4],1,1)$ se sobrepõe, pois são a mesma situação porém os meios foram feitos em ordens diferentes, que chegaram no mesmo fim, e a partir desse passo, o problema se encontra no mesmo estado, comprovando que há sobreposição de problemas.

Figura 1 – Diagrama das Chamadas Recursivas



Uma possível implementação da recursão em ***Typescript*** pode ser vista a seguir:

```
function gravar_musica(
    musicas: number[], cartuchos: number[], n: number, k: number
) {
    // se não há mais musicas ou não há mais cartuchos
    if (n < 0) return 0;

    // resultado da maior recursão
    let resultado = 0;

    // verifica a recursao para cada cartucho
    for (let i = 0; i < k; i++) {
        // se a musica não cabe no cartucho passa pro proximo
        if (cartuchos[i] - musicas[n] < 0) continue;

        let novos_cartuchos = [...cartuchos];
        novos_cartuchos[i] -= musicas[n];

        resultado = Math.max(
            resultado,
            musicas[n] + gravar_musica(
                musicas, novos_cartuchos, n - 1, k
```

```
22                  )
23              );
24          }
25
26          return Math.max(
27              // nao gravar a musica atual
28              gravar_musica(musicas, cartuchos, n - 1, k),
29              // gravar a musica atual
30              resultado
31          );
32      }
```

obs.: não foi implementado caso a caso como na formulação recursiva, mas sim uma generalização dentro do for para facilitar a interpretação do código

1.1.3 Análise

A análise se prova um pouco complexa pelo caráter dinâmico do problema, para contornar isso é necessário uma avaliação direta ao pior problema, que seria com todas as músicas cabendo nos 3 cartuchos, e outra direta ao melhor caso que seria com todas as músicas maiores a todos os cartuchos.

No pior caso, é entendido que será realizados 4 chamadas recursivas, uma para cada música em cada um dos cartuchos e outra para quando a música não for inclusa na solução, isso acontece quando todas as músicas cabem na solução. Considerando que dentro da função tudo é realizado em tempo $\Theta(1)$, pois não aumenta assintoticamente pelo aumento de músicas no problema a função recursiva é dada por:

$$T(n) = 4T(n-1) + \Theta(1) \tag{1}$$

Tudo isso para o pior dos casos, que seria quando todas as músicas cabem em todos os 3 cartuchos, ao longo de toda a recursão. Nesse caso resolvendo através do método de árvore de recursão é provado que o algorítimo é $O(4^n)$.

No melhor dos casos que seria quando nenhuma música caberia nos cartuchos, apenas uma recursão seria chamada, levando a função:

$$T(n) = T(n-1) + \Theta(1) \tag{2}$$

Fazendo uma análise rápida com a árvore de recursão, é provado que o algorítimo é limitado inferiormente por $\Omega(n)$.

1.2 Programação Dinâmica

Um algorítimo dinâmico pode ser realizado introduzindo uma forma de memoização do problema. Para isso será necessário um memo de 4 dimensões: a primeira que é indexada por n e a segunda, terceira e quarta para os possíveis valores dos cartuchos. Como apenas a memoização irá ser inclusa toda a prova de subestrutura ótima do problema se aplica na programação dinâmica também, pois as chamadas recursivas ainda se aplicam.

1.2.1 Exemplo

Por Exemplo ao escolher $mus = [7, 4, 4]$ e $cart = [8]$, na primeira chamada ($n = 2; k = 1$) irá verificar se $memo[n][cart[0]][cart[1]][cart[2]]$ já existe, caso não exista irá ser calculado para essas entradas com uma formulação semelhante a solução recursiva, se existir retornará o valor armazenado.

Isso é verificado em todas as chamadas até o caso base. Essa forma de resolver o problema traz benefícios nos casos de sobreposição de problemas, que toda a árvore de chamadas a partir do determinado estado não será calculada mais de uma vez.

Observações

Ao aplicar os três algorítimos, para o caso:

Enrtadas

8 3

7 3 3 4 4 3 2 2

9 8 9

A formulação recursiva obteve $\approx 14ms$

Enquanto o algorítimo memoizado $\approx 4ms$

E o guloso, apesar de não apresentar a solução ótima $\approx 3ms$

O algorítimo dinâmico do problema, implementada usado ***Typescript***:

```
1  const memo: number[][][][] = [];
2
3  function gravar_musica_dinamico(
4      musicas: number[], cartuchos: number[], n: number, k: number
5  ) {
6      if (n < 0 || k <= 0) return 0;
7
```

```
    // caso não esteja definido use 0 para x
    let x = cartuchos[0]??0;
    // caso não esteja definido use 0 para y
    let y = cartuchos[1]??0;
    // caso não esteja definido use 0 para z
    let z = cartuchos[2]??0;
    if (memo[n][x][y][z] != -1)
        return memo[n][x][y][z];

    let resultado = 0; // resultado da maior recursão

    // verifica a recursao para cada cartucho
    for (let i = 0; i < k; i++) {
        // se a musica não cabe no cartucho passa pro proximo
        if (cartuchos[i] - musicas[n] < 0) continue;

        let novos_cartuchos = [...cartuchos];
        novos_cartuchos[i] -= musicas[n];

        resultado = Math.max(
            resultado,
            musicas[n] + gravar_musica_dinamico(
                musicas, novos_cartuchos, n - 1, k
            )
        );
    }

    memo[n][x][y][z] = Math.max(
        // nao gravar a musica atual
        gravar_musica_dinamico(musicas, cartuchos, n - 1, k),
        // gravar a musica atual
        resultado
    );

    return memo[n][x][y][z];
}
```

obs.: Lembrando que o vetor memo deve ser inicializado antes de $checar\_numeros\_dinamico()$ ser chamada.

1.2.2 Análise

Um algorítimo de programação dinâmica pode ter uma formulação recursiva parecida com o mesmo só que não resolvido dinamicamente, porém pela propriedade de não recalcular mais de uma vez casos de sobreposição há uma redução considerável no tempo de solução do problema.

No pior dos casos o memo deverá ser preenchido em todas as dimensões, pois teria $k = 3$. Para preencher todo o memo a função teria limite superior em $O(n^4)$. No melhor caso, que seria quando há apenas 1 cartucho, a função seria limitada em $\Omega(n^2)$, pois não seria necessário preencher a terceira e quarta dimensão do memo. Generalizando o problema a função memoizada tem-se que a função é $\Theta(n^k)$.

1.3 Algorítimo Guloso

Um algoritmo guloso para esse problema pode ser um pouco grosseiro em uma primeira vista, pois há muitas possibilidades e bifurcações em suas soluções, mas algumas possibilidades de escolhas se destacam em relação a outras, por exemplo:

- Maior diferença - Escolha sempre o cartucho e a música que há maior diferença
- Menor diferença - Escolha sempre o cartucho e a música que há menor diferença
- Menores - Dado um vetor de música e cartuchos ordenados de forma decrescente pegue sempre a menor música e o menor cartucho que cabe essa música.
- Maiores - Dado um vetor de música e cartuchos ordenados pegue sempre a maior música e o maior cartucho que cabe essa música.

O decisão dos Maiores foi a que obteve melhor resultado quando comparado as outras decisões, pois o maior cartucho vai sendo decrementado progressivamente, e quando ele já não é mais viável a solução, por causa da condição dos vetores estarem ordenados, ele passa a ser o último cartucho a ser verificado.

Nota

Note que a decisão NÃO É ÓTIMA, pois há possibilidade da solução não incluir a maior música que se enquadra no maior cartucho como em:

Enrtadas

3 1

4 4 7

8

Saída

7 (ERRADO!!!)

A solução ótima desse algorítimo é 8 que seria incluir 4 + 4 na fita.

A solução gulosa do problema, implementada usado ***Typescript***:

```typescript
function gravar_musica_guloso(
    musicas: number[], cartuchos: number[], n: number, k: number
) {
    if (n < 0)
        return 0;

    let i = k;

    for (; i >= 0; i--)
        if (cartuchos[i] >= musicas[n]) break;

    if (i < 0)
        return gravar_musica_guloso(musicas, cartuchos, n - 1, k);

    // se a musica atual cabe no cartucho
    let novos_cartuchos = [...cartuchos];
    novos_cartuchos[i] -= musicas[n];

    // reordena novos_cartuchos
    while (i > 0 && novos_cartuchos[i] < novos_cartuchos[i - 1]) {
        let aux = novos_cartuchos[i];
        novos_cartuchos[i] = novos_cartuchos[i - 1];
        novos_cartuchos[i - 1] = aux;
        i--;
    }

    return musicas[n] + gravar_musica_guloso(
        musicas, novos_cartuchos, n - 1, k
    );
}
```

obs.: Lembrando que os vetores devem estar ordenados

1.3.1 Análise

A análise do algorítimo guloso é bem trivial, pois basicamente na função no pior caso todos os k cartuchos serão percorrido para todos os n cartuchos, além de ter que levar em conta que o cartucho que foi alterado deve ser reinserido em tempo $O(k)$ na lista de cartuchos, o que leva a descobrir que a recurção no pior caso é:

$$T(n) = T(n-1) + \Theta(k) \tag{3}$$

Fazendo uma análise da árvore de recursão, que terá altura n, a complexidade da função será limitada por $O(n * k)$. Inferiormente a função é limitada quando todas as músicas cabem no primeiro cartucho (maior) a ser analisado, o que leva a função recursiva, que é $\Omega(n)$:

$$T(n) = T(n-1) + \Theta(1) \tag{4}$$

2 1286 - MOTOBOY

O problema de número mil duzentos e oitenta e seis do *beecrowd*, que pode ser acessado em: https://www.beecrowd.com.br/judge/pt/problems/view/1286. Dá uma lista de entregas de pizzas com a quantidade e tempo de cada entrega, além da quantidade de pizzas que o motoboy consegue entregar. O objetivo do problema é descobrir o maior tempo de entrega que é possível formar a partir dessa lista. A primeira linha é a quantidade de entregas, a segunda a capacidade de pizzas, e as linhas subsequentes os detalhes de cada entrega. o número 0 indica fim da sequência.

Exemplo

Entradas

2

15

47 12

39 4

0

Saída

47 min.

2.1 Formulação Recursiva

Como o problema consiste em descobrir a soma da melhor combinação de entregas para serem feitas, é necessário compreender que a cada iteração é possível incluir ou não a entrega na solução. e chamar a recursão sem aquela entrega no conjunto de entregas possíveis.

- $n =$ número de entregas - 1;
- $cap =$ capacidade do entregador;
- $A =$ conjunto de entregas que podem ser feitas de [0..n];

$$f(A,n,cap) = \begin{cases} 0 & \text{se } n < 0 \text{ ou } cap = 0 \\ f(A, n-1, cap) & \text{se } cap < A[n].quantidade \\ max\ \{ & \text{se } cap >= A[n].quantidade \\ \quad f(A, n-1, cap), & \\ \quad A[n].tempo + f(A, n-1, cap - A[n].quantidade) & \\ \} & \end{cases}$$

Com a formulação recursiva, é possível identificar que quando a lista de entregas é esgotada é trivial que a função retorne 0, pois é o caso base da função. Caso o motoboy não tenha capacidade para incluir o pedido (que é analisado em $cap < A[n].quantidade$), apenas a função para o próximo pedido é chamada, sem nenhuma comparação adicional. Quando é possível adicionar o pedido, é necessário entender que é necessário pegar a maior solução seja incluindo o pedido ou sem incluir.

2.1.1 Exemplo e Subestrutura Ótima

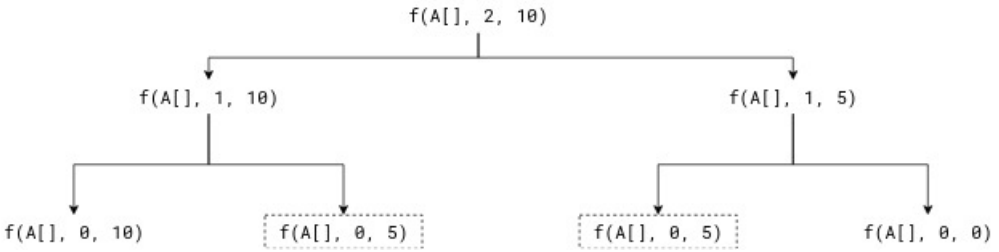
Para exemplificar o problema, será tomado $n = 2$; $cap = 10$; $A = [\{quantidade : 10, tempo : 20\}, \{quantidade : 5, tempo : 5\}, \{quantidade : 5; tempo : 10\}]$. Na primeira chamada da função será assertado que n é maior ou igual a 0, após a verificação do índice, será verificado se a entrega $A[n]$ cabe na mochila do motoboy, como 5 é menor que cap, a função chamará recursivamente tanto para caso a entrega for ignorada em: $f(A, n-1, cap)$, e para caso a entrega for incluída na solução com $f(A, n-1, cap-A[n].quantidade)$. No caso do exemplo é possível notar que é de extrema importância a chamada recursiva que ignora o pedido atual, pois a solução não inclui $n = 2$ nem $n = 1$.

Nesse caso é possível também identificar a subestrutura ótima do problema, pois devido ao caráter da função sempre pegar a chamada que retorna o maior valor, ou seja, o ótimo para o estado atual e somente incluir a entrega atual, a solução ótima e mantida para as chamadas recursivas acima dela.

2.1.2 Sobreposição de Problemas

A sobreposição de problemas ocorre nesse problema, pois na formulação recursiva descrita, há a possibilidade de combinação diferentes de pedidos gerarem a mesma capacidade, que será a entrada de uma chamada recursiva em um nível abaixo. Para sumarizar:

Figura 2 – Diagrama das Chamadas Recursivas - 1286



Uma possível implementação da recursão em ***Typescript***

```
function entrega_pizza(
    entregas: entrega[], n: number, capacidade: number
) {
    if (n < 0) return 0;

    if (capacidade < entregas[n].quantidade)
        return entrega_pizza(entregas, n - 1, capacidade);

    return Math.max(
        entregas[n].tempo + entrega_pizza(
            entregas, n - 1, capacidade - entregas[n].quantidade
        ),
        entrega_pizza(entregas, n - 1, capacidade)
    )
}
```

obs.: para o código acima e os próximos uma implementação do tipo $entrega$ foi feita como segue:

```
type entrega = {tempo: number, quantidade: number};
```

2.1.3 Análise

A recursão do problema é bem simples, por isso a análise é bem tranquila de ser feita. Note que a função não possui nenhum tipo de estrutura de repetição nem nada que custaria mais do que $\Theta(1)$ para ser realizado. Para a compreensão da complexidade do problema ao todo é necessário entender que no pior caso todas as entregas caberão na capacidade do motoboy, por isso todas chamadas serão duplas. Montando a recursão do pior caso:

$$T(n) = 2T(n-1) + \Theta(1) \tag{5}$$

Analisando a recursão, é possível chegar que o problema é assintoticamente limitado por $O(2^n)$. No melhor caso nenhuma entrega caberá na capacidade o que levará a função recursiva:

$$T(n) = T(n-1) + \Theta(1) \tag{6}$$

Com essa função recursiva é possível interpretar que a função possui limite inferior $\Omega(n)$.

2.2 Programação Dinâmica

Devido a característica de sobreposição de problemas apresentado na formulação do problema, um algorítimo usando da programação dinâmica beneficiaria muito a solução rápida do problema. Como somente a memoização irá ser inclusa no programa, toda a formulação recursiva descrita ainda será válida. Para uma boa formulação da memoização será necessário um memo bidimensional, uma para o n e outra para a capacidade cap.

É importante notar que assim como no primeiro problema, não é só porque a capacidade é uma das abrangências de uma das dimensões que a análise assintótica do problema irá alterar com base nesse valor, pois nem todos os elementos dessa dimensão é alcançável pelo conjunto de entrada.

2.2.1 Exemplo

Para exemplificar a memoização, será tomado novamente $n = 2$; $cap = 10$; $A = [\{quantidade : 10, tempo : 20\}, \{quantidade : 4, tempo : 5\}, \{quantidade : 5; tempo : 10\}]$, porém agora na primeira chamada uma verificação no memo será feita na posição $memo[2][10]$, que estará indefinido, e então a solução para esse problema será calculada e inserida no memo. Para as próximas chamadas também a verificação será feita, caso algum valor diferente esteja definido, irá usá-lo ao invés de calcular novamente.

O algorítimo dinâmico, implementado em ***Typescript***:

```
1  const memo: number[][] = [];
2
3  function entrega_pizza_dinamico(
4      entregas: entrega[], n: number, capacidade: number
5  ) {
6      if (n < 0) return 0;
7
8      if (memo[n][capacidade] >= 0)
9          return memo[n][capacidade];
10
11     let resultado = 0;
12
13     if (capacidade < entregas[n].quantidade) {
14         resultado = entrega_pizza_dinamico(
15             entregas,
16             n - 1,
17             capacidade
```

```
18            );
19        } else {
20            resultado = Math.max(
21                entregas[n].tempo + entrega_pizza_dinamico(
22                    entregas,
23                    n - 1,
24                    capacidade - entregas[n].quantidade
25                ),
26                entrega_pizza_dinamico(entregas, n - 1, capacidade)
27            );
28        }
29
30        memo[n][capacidade] = resultado;
31        return resultado;
32    }
```

2.2.2 Análise

Apesar de ter a mesma formulação recursiva, o algorítimo de programação dinâmica pode ter uma complexidade bem inferior, por reaproveitar recursões que já foram calculadas. No pior dos casos, que novamente seria quando todas as entregas cabem na capacidade do motoboy, A função teria que percorrer por todos os elementos do conjunto de entregas, chamar para quando incluir a entrega na solução e outra para quando não incluir, ou seja uma solução para cada n geraria 2 "sub soluções"o que seria necessário preencher n casas das duas dimensões do memo o que levaria a função ser limitada superiormente por $O(n^2)$.

Inferiormente a função seria limitada quando nenhuma entrega caberia na capacidade do motoboy, nesse caso a capacidade não seria alterada, então o acesso: $memo[n][CAP]$ na segunda dimensão seria uma constante CAP, o que levaria a ser necessário apenas o preenchimento da primeira dimensão n, levando ao limite inferior $\Omega(n)$.

2.3 Algorítimo Guloso

Para realizar o algorítimo guloso, primeiro é necessário selecionar a escolha gulosa do problema, nesse caso será utilizado a interpretação de que dado um vetor de entregas ordenado pela duração, será selecionado a maior entrega que cabe na capacidade atual.

2.3.1 Exemplo

O algorítimo selecionado, não necessariamente traz a solução ótima, para exemplificar isso trate $n = 2$; $cap = 10$; $A = [\{quantidade : 5, tempo : 10\}, \{quantidade : 4, tempo : 15\}, \{quantidade : 10, tempo : 20\}]$. Com esse conjunto de dados de entradas a solução retorná $20min.$ que não é a solução ótima. A solução ótima nesse caso seria $25min.$. Na primeira iteração a entrega de tempo 20 será adicionado e 10 será deduzido da quantidade restante, e então nas chamadas recursivas não haveria mais espaços para as próximas entradas.

A solução gulosa desse problema implementada em ***Typescript***:

```
function entrega_pizza_guloso(
    entregas: entrega[], n: number, capacidade: number
) {
    let i = n;
    for (; i >= 0; i--)
        // se cabe na capacidade atual
        if (capacidade > entregas[i].quantidade) break;

    // se nenhum cabe retorna 0
    if (i < 0) return 0;

    return (
        entregas[i].tempo + entrega_pizza_guloso(
            entregas, i - 1, capacidade - entregas[i].quantidade
        )
    );
}
```

2.3.2 Análise

A análise do problema é bem simples, pois só há uma chamada recursiva no problema, e a complexidade de execução interna da função é $O(n)$; $\Omega(1)$, devido ao laço de repetição iterando em todas as entregas que não cabem na solução atual. Ao realizar uma análise da recurção, no pior caso, a seguinte equação é obtida:

$$T(n) = T(n-1) + O(1) \tag{7}$$

Observe, que no pior caso, quanto mais chamadas recursivas acontecerem menos elementos serão iterados no laço for, pois ele interage diretamente com o n que determina o sub arranjo $A[0..n]$ que é levando em consideração na solução do problema. Com isso em mente é possível concluir que a função possui limite $O(n)$ no pior caso e $\Omega(n)$, por causa do laço. Com isso é possível provar que a função é resolvida em tempo $\Theta(n)$.