Universidade de São Paulo
Instituto de Matemática e Estatística
Aluna: Nara Avila Moraes
Nro. USP: 5716734

2o. semestre, 2025
Exercício 01 - 17/08

MAC5722/IME: Complexidade Computacional.
Prof.: Benjamin Merlin Bumpus

Questão 1. Arora, Barak - Capítulo 01:

(1.1) Seja f a função de *adição* que mapeia a representação de um par de números x, y para a representação do número $x + y$. Seja g a função de *multiplicação* que mapeia (x, y) para $\lfloor x \cdot y \rfloor$. Prove que tanto f quanto g são computáveis, escrevendo uma descrição completa (incluindo os estados, o alfabeto e a função de transição) das máquinas de Turing correspondentes.

(1.7) Defina uma *máquina de Turing bidimensional* como uma TM onde cada uma de suas fitas é uma grade infinita (e a máquina pode se mover não apenas para a Esquerda e Direita, mas também para Cima e para Baixo). Mostre que para toda função (tempo-construível) $T : \mathbb{N} \to \mathbb{N}$ e toda função Booleana f, se g pode ser computada em tempo $T(n)$ usando uma TM bidimensional, então $f \in \mathbf{DTIME}(T(n)^2)$.

(1.14) Prove que as seguintes linguagens/problemas de decisão em grafos estão em **P**. (Você pode escolher a representação por matriz de adjacência ou lista de adjacência para os grafos; isso não fará diferença. Consegue ver por quê?)

(a) **CONNECTED**—O conjunto de todos os grafos conexos. Ou seja, $G \in$ **CONNECTED** se todo par de vértices u, v em G está conectado por um caminho.

(b) **TRIANGLEFREE**—O conjunto de todos os grafos que não contêm um triângulo (i.e., uma tripla u, v, w de vértices distintos e conectados).

(c) **BIPARTITE**—O conjunto de todos os grafos bipartidos. Ou seja, $G \in$ **BIPARTITE** se os vértices de G podem ser particionados em dois conjuntos A, B tal que todas as arestas em G são de um vértice em A para um vértice em B (não há aresta entre dois membros de A ou dois membros de B).

(d) **TREE**—O conjunto de todas as árvores. Um grafo G é uma *árvore* se for conexo e não contiver ciclos. Equivalentemente, um grafo G é uma árvore se todo par de vértices distintos u, v em G estiver conectado por exatamente um caminho simples (um caminho é simples se não tiver vértices repetidos).

Questão 2. Arora, Barak - Capítulo 01:

(2) Descreva a prova do Teorema 1.9 em Arora, Barak em *suas próprias palavras*. Explique a intuição e dê detalhes da prova. Os pontos serão atribuídos com base na capacidade de mostrar que o conteúdo foi compreendido.

Resposta para o Ítem 1.1

Seja a descrição da Máquina de Turim de Adição e Multiplicação sua prova construtiva, seguem a descrição formal de tais máquinas, na qual a descrição das transições da máquina de multiplicação foi apresentada de forma simplificada, para evidenciar a clareza do algoritmo em vez da complexidade:

Máquina de Turing para Adição: $f(x, y) = x + y$

O objetivo é transformar uma fita inicial contendo 1^x01^y em uma fita final contendo 1^{x+y}. O algoritmo é:
1. Encontrar o separador '0' e mudá-lo para '1'.
2. Voltar ao início da fita.
3. Apagar o primeiro '1'.
4. Ir para o estado halt.

- **Alfabeto da Fita (Γ):** $\{1, 0, B\}$, onde B é o símbolo de branco.
- **Estados (Q):** $\{q_0, q_1, q_2, q_3, q_{halt}\}$
  - q_0: Estado inicial. Procura o separador '0', movendo para a direita.

- q_1: ‘0‘ encontrado e substituído. Move para a esquerda em direção ao início da fita.
- q_2: Início da fita encontrado. Move um passo para a direita para apagar o primeiro ‘1‘.
- q_3: Primeiro ‘1‘ apagado. Entra no estado de parada.
- q_{halt}: Estado final/de parada.

A função $\delta : Q \times \Gamma \to Q \times \Gamma \times \{E, D\}$ é definida como:

- **Estado q_0 (Procurar ‘0‘):**
- $\delta(q_0, 1) = (q_0, 1, D)$
- $\delta(q_0, 0) = (q_1, 1, E)$
- **Estado q_1 (Voltar ao início):**
- $\delta(q_1, 1) = (q_1, 1, E)$
- $\delta(q_1, B) = (q_2, B, D)$
- **Estado q_2 (Apagar primeiro ‘1‘):**
- $\delta(q_2, 1) = (q_3, B, D)$
- $\delta(q_2, B) = (q_{halt}, B, D)$ *(Caso de entrada x=0 e y=0)*
- **Estado q_3 (Finalização):**
- $\delta(q_3, 1) = (q_{halt}, 1, D)$
- $\delta(q_3, 0) = (q_{halt}, 0, D)$
- $\delta(q_3, B) = (q_{halt}, B, D)$

Máquina de Turing para Multiplicação

O objetivo é transformar uma fita inicial contendo 1^x01^y em uma fita final contendo 1^{xy}. O algoritmo é:

1. Para cada ‘1‘ no bloco de ‘x‘:
(a.) Marcar o ‘1‘ com um ‘X‘.
(b.) Executar uma sub-rotina para copiar todo o bloco de ‘y‘ para a área de resultado no final da fita.
2. Quando o loop principal terminar (ao não encontrar mais ‘1‘s para marcar no bloco ‘x‘), iniciar a fase de limpeza.
3. Percorrer a fita da direita para a esquerda, apagando a entrada original (os ‘X‘s, o ‘0‘ e o bloco ‘y‘), deixando apenas o bloco de resultado que foi construído.
4. Posicionar a cabeça de leitura no início do resultado.
5. Ir para o estado halt.

- **Alfabeto da Fita (Γ):** $\{1, 0, B, X, Y\}$, onde X e Y são marcadores.
- **Estados (Q):** $\{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_8, q_{halt}\}$
  - q_0: **Início do laço principal.** Procura o próximo ‘1‘ no bloco **x** para processar.
  - q_1: Marca um ‘1‘ de **x** com ‘X‘ e move para o início do bloco **y**.
  - q_2: **Início do laço interno.** Procura o próximo ‘1‘ no bloco **y** para copiar.
  - q_3: Marca um ‘1‘ de **y** com ‘Y‘ e move para a direita para a área de resultado.
  - q_4: Escreve um ‘1‘ no final da fita e inicia o retorno.
  - q_5: Retorna para a esquerda, procurando o marcador ‘Y‘.

- q_6: Marcador ‘Y‘ encontrado. Continua o laço interno para copiar o resto de **y**.
- q_7: Fim do laço interno (todo o bloco **y** foi copiado). Restaura os ‘Y‘ para ‘1‘.
- q_8: **Limpeza final.** Apaga a entrada original (x, y e os marcadores).
- q_{halt}: Estado final/de parada.

Função de Transição (δ)

A função $\delta : Q \times \Gamma \to Q \times \Gamma \times \{E, D\}$ é definida pelas seguintes transições principais:

- **Estado q_0 (Laço Principal - Procurar ‘1‘ em ‘x‘):**
- $\delta(q_0, 1) = (q_1, X, D)$ *(Encontrou um ‘1‘ em x, marca com ‘X‘ e inicia a cópia de y)*
- $\delta(q_0, X) = (q_0, X, D)$ *(Pula ‘1‘s já processados)*
- $\delta(q_0, 0) = (q_8, B, E)$ *(Não há mais ‘1‘s em x, vai para a limpeza da fita)*
- **Estado q_1 (Mover para o início de ‘y‘):**
- $\delta(q_1, 1) = (q_1, 1, D)$
- $\delta(q_1, X) = (q_1, X, D)$
- $\delta(q_1, 0) = (q_2, 0, D)$ *(Chegou em y, inicia a cópia)*
- **Estado q_2 (Laço Interno - Procurar ‘1‘ em ‘y‘):**
- $\delta(q_2, 1) = (q_3, Y, D)$ *(Marca ‘1‘ de y com ‘Y‘ e vai para o resultado)*
- $\delta(q_2, Y) = (q_2, Y, D)$
- $\delta(q_2, 0) = (q_2, 0, D)$ *(Atravessa o separador do resultado)*
- $\delta(q_2, B) = (q_7, B, E)$ *(Fim de y, a cópia está completa)*
- **Estado q_3 (Mover para o fim da fita):**
- $\delta(q_3, 1) = (q_3, 1, D)$
- $\delta(q_3, Y) = (q_3, Y, D)$
- $\delta(q_3, 0) = (q_3, 0, D)$
- $\delta(q_3, B) = (q_4, 1, E)$ *(Escreve ‘1‘ no resultado)*
- **Estado q_4 (Retornar para o marcador ‘Y‘):**
- $\delta(q_4, 1) = (q_4, 1, E)$
- $\delta(q_4, 0) = (q_4, 0, E)$
- $\delta(q_4, Y) = (q_5, Y, E)$ *(Encontrou ‘Y‘, continua o retorno)*
- **Estado q_5 (Posicionar para continuar cópia de ‘y‘):**
- $\delta(q_5, 1) = (q_5, 1, E)$
- $\delta(q_5, 0) = (q_6, 0, D)$ *(Retornou ao início de y)*
- **Estado q_6 (Restaurar ‘Y‘ e continuar laço interno):**
- $\delta(q_6, Y) = (q_2, 1, D)$ *(Restaura ‘Y‘ para ‘1‘ e procura o próximo ‘1‘ em y)*
- **Estado q_7 (Restaurar ‘y‘ e voltar para ‘x‘):**
- $\delta(q_7, 1) = (q_7, 1, E)$

- $\delta(q_7, Y) = (q_7, 1, E)$ *(Restaura todos os 'Y' para '1')*
- $\delta(q_7, 0) = (q_7, 0, E)$
- $\delta(q_7, X) = (q_0, X, D)$ *(Retorna ao laço principal para processar o próximo '1' de x)*
- **Estado q_8 (Limpeza Final):**
- $\delta(q_8, 1) = (q_8, B, E)$
- $\delta(q_8, X) = (q_8, B, E)$
- $\delta(q_8, 0) = (q_8, B, E)$
- $\delta(q_8, B) = (q_{halt}, B, D)$ *(A máquina pára posicionada no início do resultado)*

Resposta para o Ítem 1.7

O objetivo é provar que qualquer função computável em tempo $T(n)$ por uma Máquina de Turing bidimensional (2D-TM) pode ser computada em tempo $O(T(n)^2)$ por uma Máquina de Turing padrão (com fita linear de 1 dimensão). Isso demonstrará que a classe de problemas resolvidos pela 2D-TM em tempo $T(n)$ está contida em DTIME$(T(n)^2)$.

A prova é construtiva: descreveremos como uma TM padrão (usando múltiplas fitas para facilitar, o que é sabidamente equivalente em poder a uma TM de fita única com sobrecarga polinomial) pode simular uma 2D-TM.

Passo 1: Representando a Fita 2D em uma Fita 1D

O desafio inicial é mapear a grade infinita bidimensional para uma fita linear. A observação fundamental é que, em $T(n)$ passos, a 2D-TM só consegue visitar células (x, y) que estão a uma distância Manhattan máxima de $T(n)$ da origem $(0, 0)$. Todo o cômputo ocorre, portanto, dentro de um quadrado de dimensões $(2T(n) + 1) \times (2T(n) + 1)$.

Para mapear eficientemente essa área finita da grade, utilizaremos um **mapeamento em serpentina (ou zigue-zague)**. As linhas da grade são armazenadas sequencialmente na fita 1D, mas com direções alternadas para otimizar a adjacência vertical.

- Linhas com índice y par são escritas da esquerda para a direita.
- Linhas com índice y ímpar são escritas da direita para a esquerda.

Usaremos um símbolo especial, como `#`, para delimitar o início e o fim de cada linha.

Exemplo de Layout na Fita 1D:

```
... # c(-1,-1) c(-1,0) c(-1,1) # c(0,1) c(0,0) c(0,-1) # c(1,-1) c(1,0) c(1,1) # ...
```

Onde $c(x, y)$ representa o conteúdo da célula na coordenada (x, y).

Passo 2: A Simulação

A simulação será realizada por uma TM padrão com 2 fitas:

1. **Fita 1 (Fita de Dados):** Armazena a grade 2D usando o mapeamento em serpentina. O cabeçote desta fita sempre aponta para a célula que corresponde à posição atual do cabeçote da 2D-TM.
2. **Fita 2 (Fita de Posição):** Uma fita auxiliar que armazena a coordenada y (o número da linha) da posição atual, para facilitar a navegação vertical.

A simulação de um único passo da 2D-TM consiste em:

1. **Leitura:** A TM simuladora lê o símbolo sob seu cabeçote na Fita 1.
2. **Decisão:** Com base no símbolo lido e no estado atual da 2D-TM (armazenado internamente), a simuladora consulta a função de transição da 2D-TM para obter o novo símbolo, o novo estado e a direção do movimento (Cima, Baixo, Esquerda, Direita).
3. **Escrita:** O novo símbolo é escrito na Fita 1, sobre o símbolo antigo.
4. **Movimento:** O cabeçote da Fita 1 é movido para simular o movimento da 2D-TM.

Passo 3: Análise de Custo do Movimento

A complexidade da simulação de um passo é dominada pelo custo do movimento.

- **Movimento para Esquerda/Direita:** Um movimento horizontal na grade 2D corresponde, na maioria dos casos, a um movimento de uma única célula na Fita 1, devido ao mapeamento em serpentina. O custo é, portanto, $O(1)$.
- **Movimento para Cima/Baixo:** Este é o caso mais custoso. Para mover de (x, y) para $(x, y+1)$ (Cima), o cabeçote da Fita 1 deve saltar do bloco da linha y para o bloco da linha $y+1$. Isso exige que a TM simuladora:
  1. Percorra a linha atual até encontrar o marcador `#`.
  2. Percorra a próxima linha até encontrar a célula na coluna correspondente.

  Como a 2D-TM executa $T(n)$ passos, a dimensão máxima de qualquer linha é $2T(n)+1$. A distância a ser percorrida na Fita 1 é, portanto, proporcional ao comprimento de uma linha. O custo de um movimento vertical é $O(T(n))$.

Passo 4: Custo Total da Simulação

O custo para simular cada passo da 2D-TM é limitado pelo movimento mais caro, o vertical.

- Custo para simular 1 passo da 2D-TM: $O(T(n))$.
- Número total de passos a simular: $T(n)$.

O tempo total de simulação na TM padrão é o produto desses dois fatores:

$$T_{\text{sim}}(n) = (\text{Número de passos da 2D-TM}) \times (\text{Custo máximo por passo})$$

$$T_{\text{sim}}(n) = T(n) \times O(T(n)) = O(T(n)^2)$$

Conclusão

Demonstramos que uma 2D-TM operando em tempo $T(n)$ pode ser simulada por uma TM padrão em tempo $O(T(n)^2)$. Consequentemente, qualquer função f computável nesse regime pela 2D-TM pertence à classe de complexidade de tempo determinístico DTIME$(T(n)^2)$. A premissa de que $T(n)$ é uma função tempo-construível é necessária formalmente para que a TM simuladora possa, por exemplo, delimitar o espaço de fita necessário ou utilizar um "relógio" para controlar o tempo de simulação.

Resposta para o Ítem 1.14

Prova de que Problemas de Grafos Pertencem à Classe P

A classe **P** contém todos os problemas de decisão que podem ser resolvidos por uma máquina de Turing determinística em tempo polinomial em relação ao tamanho da entrada. Para provar que um problema está em **P**, basta apresentar um algoritmo que o resolva em tempo polinomial.

Nos casos a seguir, vamos considerar um grafo $G = (V, E)$ com $n = |V|$ vértices e $m = |E|$ arestas.

a) CONNECTED

Problema

Determinar se um grafo G é conexo. Um grafo é conexo se para todo par de vértices $u, v \in V$, existe um caminho entre u e v.

Prova

Podemos determinar se um grafo é conexo usando um algoritmo de busca em largura (BFS) ou busca em profundidade (DFS).

Algoritmo

1. Escolha um vértice inicial arbitrário $s \in V$.
2. Inicie uma busca (BFS ou DFS) a partir de s.
3. Mantenha um contador de vértices visitados.
4. Após a busca terminar, verifique se o número de vértices visitados é igual a n (o número total de vértices no grafo).
5. Se todos os n vértices foram visitados, o grafo é conexo. Caso contrário, não é.

Análise de Complexidade

Tanto a busca em largura (BFS) quanto a busca em profundidade (DFS) têm uma complexidade de tempo de $O(n + m)$ quando o grafo é representado por uma lista de adjacência. Se for usada uma matriz de adjacência, a complexidade é $O(n^2)$. Ambos os tempos, $O(n+m)$ e $O(n^2)$, são polinomiais em relação ao tamanho da entrada.

Portanto, **CONNECTED** $\in$ **P**.

b) TRIANGLEFREE

Problema

Determinar se um grafo G não contém um triângulo. Um triângulo é um conjunto de três vértices distintos $\{u, v, w\}$ tal que todos são mutuamente adjacentes.

Prova

Podemos verificar a ausência de triângulos iterando por todas as combinações possíveis de três vértices.

Algoritmo

1. Itere por todos os trios de vértices distintos $\{u, v, w\}$ do grafo.
2. Para cada trio, verifique se as arestas (u, v), (v, w) e (u, w) existem em G.
3. Se tal trio for encontrado, o grafo contém um triângulo, então ele **não** está em TRIANGLEFREE. O algoritmo pode parar e retornar "Não".
4. Se o algoritmo verificar todos os trios e não encontrar nenhum triângulo, o grafo está em TRIANGLEFREE. O algoritmo retorna "Sim".

Análise de Complexidade

O número de trios de vértices distintos em um grafo com n vértices é $\binom{n}{3} = \frac{n(n-1)(n-2)}{6}$, que é $O(n^3)$. Para cada trio, a verificação da existência das três arestas leva um tempo constante, $O(1)$, se o grafo for representado por uma matriz de adjacência. Portanto, a complexidade total do algoritmo é $O(n^3)$.

Como $O(n^3)$ é um tempo polinomial, **TRIANGLEFREE** $\in$ **P**.

c) BIPARTITE

Problema

Determinar se um grafo G é bipartido. Um grafo é bipartido se seus vértices podem ser divididos em dois conjuntos disjuntos, A e B, tal que toda aresta conecta um vértice em A a um vértice em B.

Prova

Um grafo é bipartido se, e somente se, ele não contém ciclos de comprimento ímpar. Podemos verificar isso usando um algoritmo de coloração com duas cores, geralmente implementado com uma busca em largura (BFS).

Algoritmo

1. Inicialize um array de cores `cor[1...n]` com um valor nulo (e.g., 0) para todos os vértices.
2. Para cada vértice $v \in V$:
   (a) Se `cor[v]` ainda for nulo, inicie uma BFS a partir de v:
   (b) Pinte v com a primeira cor (e.g., 1) e coloque-o em uma fila.
   (c) Enquanto a fila não estiver vazia, retire um vértice u.
   (d) Para cada vizinho w de u:
      i. Se w não estiver colorido (`cor[w]` é nulo), pinte-o com a cor oposta a u e adicione-o à fila.
      ii. Se w já estiver colorido e tiver a mesma cor que u, o grafo contém um ciclo ímpar e, portanto, não é bipartido. O algoritmo pode parar e retornar "Não".
3. Se a busca terminar para todos os componentes do grafo sem encontrar conflitos de cor, o grafo é bipartido. Retorne "Sim".

Análise de Complexidade

Este algoritmo visita cada vértice e cada aresta no máximo uma vez, assim como uma BFS padrão. Sua complexidade de tempo é $O(n + m)$ com uma lista de adjacência, que é polinomial.

Portanto, **BIPARTITE** $\in$ **P**.

d) TREE

Problema

Determinar se um grafo G é uma árvore. Uma árvore é um grafo conexo e acíclico.

Prova

Um grafo com n vértices é uma árvore se, e somente se, ele satisfaz duas das três seguintes propriedades:

i. O grafo é conexo.

ii. O grafo não contém ciclos (é acíclico).

iii. O grafo tem exatamente $n-1$ arestas.

Podemos criar um algoritmo eficiente verificando as propriedades (i) e (iii).

Algoritmo

1. Conte o número de vértices (n) e o número de arestas (m) no grafo.
2. Verifique se $m = n - 1$. Se não for, o grafo não é uma árvore. Retorne "Não".
3. Se $m = n - 1$, verifique se o grafo é conexo usando o algoritmo da parte (a) (BFS ou DFS).
4. Se o grafo for conexo e tiver $n-1$ arestas, ele é uma árvore. Retorne "Sim". Caso contrário, retorne "Não".

Análise de Complexidade

1. Contar vértices e arestas pode ser feito em $O(n+m)$ com uma lista de adjacência.
2. A verificação da condição $m = n - 1$ é $O(1)$.
3. Verificar a conectividade, como vimos, leva $O(n+m)$ tempo.

A complexidade total é dominada pela contagem e pela verificação de conectividade, resultando em $O(n+m)$. Este tempo é polinomial.

Portanto, **TREE** $\in$ **P**.

Resposta para o Ítem 2

O **Teorema 1.9** afirma a existência de uma Máquina de Turing Universal (U) que pode simular qualquer outra Máquina de Turing M_α de forma eficiente. Especificamente, se M_α para em T passos numa entrada x, a simulação $U(x, \alpha)$ para em $O(T \log T)$ passos. A seguir, descreve-se a prova dessa afirmação, focando na intuição e nos detalhes técnicos.

Desafio

A principal dificuldade em construir uma UTM eficiente reside em simular uma máquina M com múltiplas fitas de trabalho (k fitas), cujos cabeçotes se movem de forma independente, usando uma UTM U com apenas uma fita de trabalho.

A intuição central da prova é uma inversão de perspectiva: *"Se o cabeçote não pode ir até a célula desejada, a célula virá até o cabeçote."* Em vez de mover o cabeçote de U por longas distâncias, ele permanece fixo na posição

0, e o conteúdo da fita é deslocado para debaixo dele.

Uma abordagem ingênua seria, a cada passo de M, deslocar todos os símbolos em uma das k pistas paralelas da fita de U para simular o movimento do cabeçote correspondente. No entanto, como a fita pode ter até $O(T)$ símbolos, cada passo da simulação custaria $O(T)$, levando a um tempo total ineficiente de $O(T^2)$. A prova supera esse obstáculo com uma estrutura de dados engenhosa e uma análise de custo amortizado.

A Estrutura de Dados: Fitas com Zonas

Para otimizar a operação de deslocamento (shift), cada uma das k pistas paralelas na fita de trabalho de U é organizada em **zonas** de tamanhos exponencialmente crescentes, dispostas simetricamente ao redor da posição 0 do cabeçote.

- Para cada $i \geq 0$, existem duas zonas: L_i (à esquerda) e R_i (à direita).
- O tamanho (número de células) de L_i e R_i é 2^i. Assim, temos L_0, R_0 com 1 célula cada, L_1, R_1 com 2 células cada, L_2, R_2 com 4 células cada, e assim por diante.
- Um símbolo especial de "buffer" (representado como $\otimes$) é usado para preencher espaços vazios, permitindo que os dados sejam gerenciados de forma mais flexível.

Essa estrutura é mantida por um conjunto de **invariantes** durante toda a computação:

1. Cada zona (L_i ou R_i) está em um de três estados: **vazia**, **meio-cheia** ou **cheia** de símbolos que não são de buffer.
2. A união de um par de zonas simétricas, $L_i \cup R_i$, está sempre em um de três estados: ou ambas estão vazias, ou ambas estão cheias, ou ambas estão meio-cheias.
3. A célula na posição 0, sob o cabeçote de U, sempre contém um símbolo útil (não-buffer).

Operação de Deslocamento e Análise de Custo Amortizado

Quando U precisa simular um movimento do cabeçote de M (por exemplo, para a esquerda), ela realiza uma operação de deslocamento que move todo o conteúdo da pista relevante uma posição para a direita, trazendo a célula desejada para a posição 0. A eficiência vem de como essa operação é implementada com a estrutura de zonas.

O processo funciona como um sistema de "transbordamento" em cascata. Se um símbolo precisa ser movido para a zona L_0 e esta já está cheia, seu conteúdo é "empurrado" para a zona L_1. Se L_1 também estiver cheia, o processo continua para L_2, e assim por diante, até encontrar uma zona que não esteja cheia. A operação que afeta zonas até o índice i_0 tem um custo proporcional ao tamanho total dessas zonas, que é $O(\sum_{j=0}^{i_0} 2^j) = O(2^{i_0})$.

A chave para a eficiência está na **análise amortizada**:

- Uma operação de deslocamento **cara**, com custo $O(2^i)$, só ocorre quando todas as zonas $L_0, \ldots, L_{i-1}$ (ou $R_0, \ldots, R_{i-1}$) estão cheias.
- Imediatamente após essa operação cara, essas zonas $L_0, \ldots, L_{i-1}$ ficam meio-cheias.
- Para que elas fiquem cheias novamente e permitam a ocorrência de outra operação de custo $O(2^i)$, é necessário que um total de $\sum_{j=0}^{i-1} 2^j = 2^i - 1$ símbolos sejam movidos para elas por meio de operações mais baratas (de índice $< i$).

Isso significa que cada operação cara é "paga" por um grande número de operações baratas que a precedem. Embora um único passo da simulação possa ser caro, o custo *médio* (amortizado) por passo é baixo.

Conclusão da Prova

A análise matemática formaliza essa intuição. O número de vezes que uma operação de deslocamento de índice i ocorre durante os T passos da simulação é no máximo $T/2^{i-1}$. O custo total de todas as operações de índice i é, portanto, $(T/2^{i-1}) \times O(2^i) = O(T)$.

Como o número máximo de zonas necessárias é tal que a soma de seus tamanhos cubra todos os T possíveis símbolos, o índice máximo i é da ordem de $\log T$. Somando os custos para todos os níveis de índice i de 1 a $\log T$, o trabalho total para simular os T passos de M em uma de suas fitas é:

$$\sum_{i=1}^{\log T} O(T) = O(T \log T)$$

Considerando todas as k fitas, o custo total da simulação é $O(k \cdot T \log T)$. Como k é uma constante que depende apenas da máquina M_α sendo simulada (e não da entrada x ou do tempo T), ele é absorvido pela notação O, provando que a simulação é concluída em $O(T \log T)$ passos.