DCC206 – Algoritmos 1

Aula 12 – Dividir e Conquistar

Professor Renato Vimieiro

DCC/ICEx/UFMG

Introdução

- O segundo paradigma de desenho de algoritmos que iremos estudar é o “dividir-e-conquistar”
- Enquanto a ideia central dos algoritmos gulosos era tomar decisões ótimas locais para solucionar um problema, na abordagem dividir-e-conquistar optamos por quebrar o problema em problemas menores
- Dividimos o problema em subproblemas (disjuntos) menores com a intenção de obter soluções mais simples para eles
- Então, combinamos as soluções dos subproblemas para compor a solução do problema maior
- Já estudamos pelo dois representantes desse paradigma: mergesort e o quicksort

Introdução

- Antes de iniciarmos a discussão de outros problemas que admitem soluções com essa abordagem, vamos estudar como analisar a complexidade desses algoritmos
- Vamos ver alguns métodos para deduzir a complexidade de um algoritmo baseado no paradigma “dividir-e-conquistar”
- Em particular, vamos estudar um teorema que nos permite deduzir rapidamente a complexidade de vários algoritmos, caso eles possuam certas propriedades

Conceitos básicos

- Vamos começar relembrando alguns conceitos relacionados a esses algoritmos (ou à sua complexidade)
- Vamos analisar o comportamento do mergesort para iniciarmos essa revisão
- A ideia do algoritmo é: para ordenar uma lista de elementos, dividimos a lista em duas, ordenamos cada um desses subproblemas, e combinamos as duas listas ordenadas em uma única lista
- Se considerarmos que o custo da ordenação de uma lista com n elementos é S(n), definimos que o custo da ordenação é dado pela ordenação de cada lista mais o custo de dividir e combinar as listas
- $S(n) = S(n/2) + S(n/2) + f(n)$
- Contudo, se a lista for muito pequena, menos de 2 elementos, o custo de ordenação é constante
  - $S(2)=b$
  - Vamos abusar da notação e não mostrar a função para valores de n pequenos, já que assumiremos que esse custo é constante

Conceitos básicos

- Equações (ou inequações) similares a essas são chamadas de **relações de recorrência**
- Relações de recorrência definem elementos a partir de elementos anteriores (recursivamente)
- Assim, dada a natureza dos algoritmos “dividir-e-conquistar”, elas são bastante adequadas para modelar seu tempo de execução
- No entanto, estamos mais acostumados a analisar o comportamento assintótico dos algoritmos.
- Portanto, precisamos encontrar uma forma de resolver essas relações para deduzir uma fórmula fechada para o comportamento assintótico desses algoritmos

Substituição iterativa

- O primeiro método que iremos analisar é chamado de **substituição iterativa**
- A ideia central dessa abordagem é expandir a equação com o intuito de observar um padrão e, assim, deduzir uma fórmula fechada
- Por exemplo, sabendo que temos um custo linear para combinar e dividir as listas no mergesort, obtemos a seguinte relação
  - $S(n) = 2S(n/2) + bn$
- Aplicando o método, ou seja, substituindo a definição de S() para n/2, temos
  - $S(n) = 2[2S(n/4) + bn/2] + bn = 4S(n/4) + 2bn$
- Aplicando novamente para n/4, temos
  - $S(n) = 4[2S(n/8) + bn/4] + 2bn = 8S(n/8) + 3bn$

Substituição iterativa

- Seguindo essa lógica, notamos que, após i substituições, temos
  - $S(n) = 2^iS(n/2^i) + ibn$
- Eventualmente, não poderemos mais dividir um problema em subproblemas
- Ou seja, quando $n/2^i = 1$, a solução terá custo b
- Isso ocorre quando i = log n
- Logo, nossa equação será
  - $S(n) = nb + bn \log n$,
  - Ou seja $S(n) = O(n \log n)$

Substituição iterativa

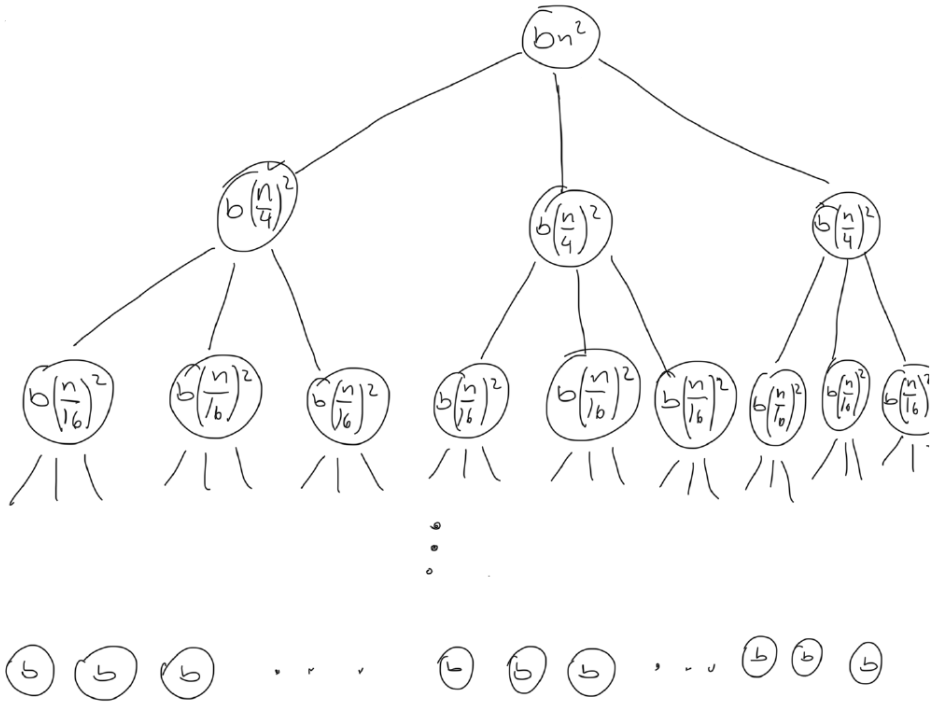
- Embora essa abordagem seja útil, ela apresenta algumas peculiaridades
- Nem sempre é fácil perceber o padrão recorrente nas substituições
- Nesses casos, precisamos demonstrar a correção da nossa interpretação de padrão
- Essa demonstração é normalmente feita por indução matemática

Árvore de recursão

- A segunda abordagem que iremos analisar tem apelo mais visual, porém é bastante relacionada à primeira
- Ao invés de fazer manipulações algébricas, como na substituição iterativa, o método baseado na árvore de recursão usa a representação visual das chamadas para deduzir o custo total
- Construímos uma árvore em que cada nó representa um subproblema (uma possível substituição)
- Adicionalmente, associamos o *overhead* necessário para criar e combinar os subproblemas naquele nó
- O custo total do algoritmo é a soma de todos os overheads dos nós dessa árvore

Árvore de recursão

- Considere o seguinte exemplo: $T(n) = 3T(n/4) + bn^2$
- Temos a seguinte árvore



Árvore de recursão

- Essa árvore tem altura h = ($\log_4$ n) + 1
- O número de folhas é $3^{\log_4 n}$
- A soma total dos custos é
  - $\left(\sum_{i=0}^{\log_4 n-1} \left(\frac{3}{16}\right)^i bn^2\right) + \left(3^{\log_4 n}\right)b$
  - $= \left(\sum_{i=0}^{\log_4 n-1} \left(\frac{3}{16}\right)^i bn^2\right) + \left(n^{\log_4 3}\right)b$
  - $< \left(\sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i bn^2\right) + \left(n^{\log_4 3}\right)b$
    - $= \left(\frac{1}{1-\frac{3}{16}} bn^2\right) + \left(n^{\log_4 3}\right)b$ = $O(n^2)$ (soma de uma pg infinita decrescente com razão menor que 1 é 1/(1-r))

O teorema mestre

- Os métodos anteriores são úteis para resolver relações de recorrência, mas exigem do usuário um certo nível de conhecimento matemático para deduzir com efetividade a complexidade de tempo
- Existe um método, baseado em um teorema (mestre), que permite resolver relações de recorrência gerais sem grandes esforços
- O teorema mestre, contudo, só é aplicado a relações de recorrência que satisfaçam as propriedades listadas nele
- Vamos considerar relações genéricas com a seguinte forma: $T(n) = aT(n/b) + f(n)$
- Ou seja, nosso algoritmo divide o problema em *a* subproblemas de tamanho *n/b* e gasta tempo f(n) para dividir/combinar as soluções

O teorema mestre

Teorema (Mestre): seja T(n) = aT(n/b) + f(n), em que a ≥ 1, b > 1 são constantes reais, e f(n) uma função assintoticamente positiva.

1. Se existe uma constante r > 0 tal que f(n) = O($n^{(\log_b a)-r}$), então T(n) = Θ($n^{(\log_b a)}$)
2. Se f(n) = Θ($n^{(\log_b a)}$), então T(n) = Θ($n^{(\log_b a)} lg\ n$)
3. Se f(n) = Ω($n^{(\log_b a)+r}$), para r>0, e af(n/b) ≤ cf(n) para uma constante c < 1 e n grande, então T(n) = Θ(f(n))

O teorema mestre

- Antes de vermos exemplos da aplicação do teorema, vamos entender o que ele diz
- O caso 1 diz que se a função f é polinomialmente menor que $n^{(\log_b a)}$, então o custo da resolução é dominado pelo custo dos problemas individuais
- O caso 2 diz que se a função f(n) é assintoticamente equivalente a $n^{(\log_b a)}$, então o custo da resolução é o da função “aplicada nos diferentes níveis da árvore”
- O caso 3 diz que se função f(n) for polinomialmente maior que $n^{(\log_b a)}$, então ela domina o custo da resolução
- Note que os 3 casos não são exaustivos. A função f(n) pode, por exemplo, ser menor que $n^{(\log_b a)}$, mas não polinomialmente menor.
- Nesses casos, o teorema não pode ser utilizado

Exemplos de uso do teorema

- $T(n) = 9T(n/3) + n$
- $a=9$, $b=3$ e $f(n) = n$, $n^{(\log_3 9)} = n^2$
- $f(n) = O(n^{2-r})$ onde $r=1$
- Logo, $T(n) = \Theta(n^2)$

Exemplos de uso do teorema

- T(n) = T(2n/3) + 1
- a = 1, b=3/2, f(n) = 1, $n^{(\log_{3/2} 1)} = 1$
- O caso 2 se aplica, já que f(n) = Θ(1)
- Portanto, T(n) = Θ(lg n)

Exemplos de uso do teorema

- $T(n) = 9T(n/3) + n^{2.5}$
- $a=9, b=3, f(n)= n^{2.5}$, $n^{(\log_3 9)} = n^2$
- Nesse exemplo, o caso 3 se aplica com $r=0.5$, $f(n) = \Omega(n^{2.5})$
- Portanto, $T(n) = \Theta(n^{2.5})$

Exemplo de uso do teorema

- $T(n) = 2T(\sqrt{n}) + \log n$
- Esse exemplo não parece obedecer às propriedades do teorema
- Contudo, se tomarmos $k = \log n$, podemos reescrever a função como
  - $T(n) = T(2^k) = 2T(2^{k/2}) + k$
- Agora, fazendo $S(k) = T(2^k)$, obtemos
  - $S(k) = 2S(k/2) + k$
  - $a=2$, $b=2$, $f(k)=k$, $k^{(\log_2 2)} = k$
  - Logo, $f(k) = \Theta(k)$ e $S(k) = \Theta(k \log k)$
  - Assim, $T(n) = \Theta(\log n \log \log n)$

Leitura

- Seção 4.5 CRLS
- Seção 11.1 Goodrich e Tamassia