Divisão e Conquista

prof. Fábio Luiz Usberti

MC521 - Desafios de Programação I

Instituto de Computação - UNICAMP - 2018

Sumário

Divisão e conquista

Introdução

- O método de divisão e conquista é um paradigma de solução de problemas onde é possível tornar um problema mais simples dividindo-o em partes menores para depois conquistá-las individualmente. Os passos desse método incluem:

1. Divisão: reduzir o problema original em um ou mais subproblemas.
2. Conquista: obter as soluções para os subproblemas.
3. Combinação: combinar as soluções dos subproblemas para obter uma solução do problema original.

- Exemplos: Quicksort, Mergesort, Árvore de Segmentos, Busca binária.

Cálculo de potências

- Desejamos calcular x^p, tal que a potência p é um inteiro positivo. É possível tratar esse problema por divisão e conquista da seguinte forma:

1. Sabemos resolver o caso base $p = 0$: $x^0 = 1$.
2. Consideramos por hipótese (de indução) que sabemos resolver o subproblema x^{p-1}.
3. O problema original pode então ser reescrito como:

$$x^p = \begin{cases} 1 & \text{se } p = 0 \\ x \cdot x^{p-1} & \text{caso contrário} \end{cases}$$

Divisão e conquista

Cálculo de potências

- Abaixo é fornecida uma implementação recursiva da estratégia de divisão e conquista proposta:

```
long potencia(long x, long p){
   if (p == 0)
      return 1;
   else
      return x*potencia(x,p-1);
}
```

- Qual a complexidade do algoritmo acima?

Cálculo de potências

- Enquanto a primeira abordagem requer tempo $O(n)$, existe uma estratégia alternativa de divisão e conquista mais eficiente.

1. Sabemos resolver o caso base $p = 0$: $x^0 = 1$.
2. Fortalecemos a hipótese (de indução): agora sabemos resolver o subproblema x^k para qualquer $1 \leqslant k < p$.
3. O problema original pode então ser reescrito como:

$$x^p = \begin{cases} 1 & p = 0 \\ x^{\frac{p}{2}} \cdot x^{\frac{p}{2}} & p \text{ é par} \\ x \cdot x^{\frac{p-1}{2}} \cdot x^{\frac{p-1}{2}} & p \text{ é ímpar} \end{cases}$$

Cálculo de potências

- A segunda estratégia de divisão e conquista pode ser implementada da seguinte forma:

```
long potencia2(long x, long p){
   long aux;
   if (p == 0)
      return 1;
   else if (p % 2 == 0) {
      aux = potencia(x,p/2);
      return aux*aux;
   } else {
      aux = potencia(x,(p-1)/2);
      return x*aux*aux;
   }
}
```

- Obs 1: Cada chamada recursiva do algoritmo reduz o valor da potência pela metade. Portanto, a quantidade de chamadas recursivas é limitada por $O(\log(n))$.
- Obs 2: O uso da variável auxiliar para armazenar o valor do subproblema é essencial para a eficiência do algoritmo.

Busca binária

Uso canônico

- O uso canônico de busca binária consiste em encontrar um elemento em um vetor ordenado. Verifica-se a mediana do vetor em busca do elemento desejado. Se não encontrar e ainda houver outros elementos não explorados, continua a busca no subvetor (esquerdo ou direito) onde o elemento tem chances de estar.
- Como o espaço de busca é dividido à metade a cada iteração do algoritmo, sua complexidade é limitada por $O(\log n)$.
- A biblioteca C++ STL contém métodos que implementam busca binária. São eles: (`lower_bound`, `upper_bound`, `binary_search` da classe `algorithm`).

Método da bisecção

Raiz de funções

- O método da bisecção utiliza o mesmo princípio da busca binária para encontrar zeros de funções contínuas que podem ser difíceis de calcular analiticamente.
- Problema: Considere um veículo comprado com empréstimo cujas parcelas de valor d devem ser pagas durante m meses. Suponha que o valor original do carro seja v e que o banco cobra um juros de i % ao mês para o montante que falta ser pago. Qual deve ser a quantia d a ser paga (com precisão de duas casas decimais)?

Método da bisecção

Raiz de funções

- Supondo $d = 576.19$, $m = 2$, $v = 1000$ e $i = 10\%$.
- Após o primeiro mês, a dívida passa a ser $1000 \times 1.1 - 576.19 = 523.81$.
- Após o segundo mês, a dívida passa a ser $523.81 \times 1.1 - 576.19 \approx 0$.
- Considere a função $saldo(d)$ (para m, v, i constantes) que retorna o saldo da dívida e para a qual deseja-se obter a raiz d_0 tal que $saldo(d_0) = 0$.
- Se for possível determinar analiticamente essa função, o problema pode ser resolvido de forma exata.
- Se o cálculo analítico não for trivial, uma solução consiste em implementar o método da bisecção.

```
double saldo(double d){
   // supondo M, V, I parametros constantes
   int mes;
   double valor = V;
   for (mes=1; mes<=M; mes++) {
      valor = valor*(1.0+I) - d;
   }
   return valor;
}
```

Método da bisecção

Raiz de funções

- Como entrada do método, escolhe-se um intervalo de busca inicial $[a..b]$, tal que $f(a)$ e $f(b)$ tenham sinais opostos, ou seja, há uma raiz nesse intervalo.
- No exemplo, é possível escolher $a = 0.01$ e $b = (1 + i)v$.
- O número de iterações do método da bisecção é limitado por $\log \frac{b-a}{\epsilon}$, onde ϵ é a precisão desejada.
- Para o exemplo tem-se um número de iterações $\log \frac{1099.99}{\epsilon}$. Mesmo utilizando uma precisão de nove casas decimais ($\epsilon = 10^{-9}$), o método executará menos que 40 iterações.

```
double bisseccao(double a, double b){

   double valor = saldo((a+b)/2);
   if (fabs(valor) < EPS)
      return (a+b)/2;
   else if (valor > 0.0)
      return bisseccao(a,(a+b)/2);
   return bisseccao((a+b)/2,b);
}
```

Raiz de funções

a	**b**	**d** $= \frac{a+b}{2}$	**status:** $f(d, m, v, i)$	**action**
0.01	1100.00	550.005	undershoot by 54.9895	increase d
550.005	1100.00	825.0025	overshoot by 522.50525	decrease d
550.005	825.0025	687.50375	overshoot by 233.757875	decrease d
550.005	687.50375	618.754375	overshoot by 89.384187	decrease d
550.005	618.754375	584.379688	overshoot by 17.197344	decrease d
550.005	584.379688	567.192344	undershoot by 18.896078	increase d
567.192344	584.379688	575.786016	undershoot by 0.849366	increase d
...	...	...	a **few** iterations later ...	...
...	...	576.190476	stop; error is now less than ϵ	answer = 576.19

Busca binária na resposta

Dividindo o espaço de busca

- Simplificadamente, a estratégia de busca binária na resposta consiste em uma estratégia para reduzir o espaço de busca de problemas com as seguintes características:
- São problemas cujas soluções apresentam veredítos binários como “sim” ou “não”, “verdadeiro” ou “falso”, “viável” ou “inviável”.
- Dada uma solução x para o problema P, se $P(x)$ resultar em “sim”, então $P(x')$ resultará em “sim” para $x' \geqslant x$. De modo análogo, se $P(x)$ resultar em “não”, então $P(x'')$ resultará em “não” para $x'' \leqslant x$.
- Em outras palavras, se os vereditos de todas as possíveis entradas (em ordem) do problema fossem impressas, o resultado seria uma sequência de “não” seguida por uma sequência de “sim”.

Busca binária na resposta

Dividindo o espaço de busca

- Problema (UVa 11935): Considere uma pessoa atravessando uma estrada de carro. O veículo possui um tanque inicialmente cheio de combustível. No caminho podem ocorrer os seguintes eventos:

1. Digirir – consome combustível.
2. Vazamento de combustível – consome combustível.
3. Encontrar posto de abastecimento – repõe o tanque de combustível em sua capacidade máxima.
4. Encontrar mecânico – conserta todos os vazamentos.
5. Atingir destino – final da viagem.

- Desejamos determinar a capacidade mínima do tanque de combustível de modo que o veículo consiga alcançar seu destino. A resposta deve apresentar uma precisão de três casas decimais.

Dividindo o espaço de busca

- A partir da capacidade do veículo, é possível simular os eventos na ordem em que ocorrem e determinar se o veículo atinge ou não seu destino. O problema reside no fato de que não sabemos a capacidade, pois é exatamente essa variável que desejamos determinar.
- Uma maneira ingênua de resolver esse problema consiste em testar todos os possíveis valores para a capacidade do tanque $[0.000, 10000.000]$, mas isso corresponde a $10M$ de simulações, o que resultará no veredito TLE.
- Uma propriedade do problema a ser explorada pelo método da bisecção consiste no fato de que se uma capacidade X não é suficiente para o veículo completar sua viagem, então nenhuma capacidade no intervalo $[0.000, X - 0.001]$ será viável. Logo, a solução estará no intervalo complementar $[X, 10000.000]$.

Dividindo o espaço de busca

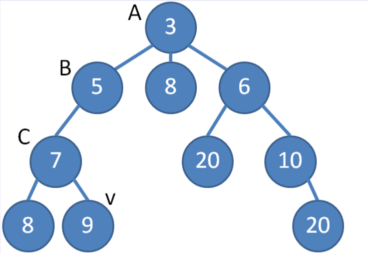
```
#define EPS 1e-9 // this value is adjustable; 1e-9 is usually small enough
bool can(double f) { // details of this simulation is omitted
    // return true if the jeep can reach goal state with fuel tank capacity f
    // return false otherwise
}
// inside int main()
// binary search the answer, then simulate
double lo = 0.0, hi = 10000.0, mid = 0.0, ans = 0.0;
while (fabs(hi - lo) > EPS) { // when the answer is not found yet
   mid = (lo + hi) / 2.0; // try the middle value
   if (can(mid)) { ans = mid; hi = mid; } // save the value, then continue
   else lo = mid;
}
printf("%.3lf\n", ans); // after the loop is over, we have the answer
```

Problema “My Ancestor”

- Problema “My Ancestor”: Considere uma árvore ponderada (peso nos vértices) com $N \leqslant 80000$ vértices e que mantém a propriedade de heap de mínimo, ou seja, os pesos nos vértices aumentam da raiz até as folhas. Dado um vértice v e um peso P, deseja-se encontrar o vértice mais próximo da raiz, ancestral de v, que tenha peso pelo menos P. Em uma mesma árvore poderão ser realizadas $Q \leqslant 20000$ consultas “off-line”.
- Uma solução ingênua consiste em realizar uma busca linear $O(n)$, começando pelo vértice v e movendo-se em direção à raiz da árvore até encontrar um ancestral cujo peso seja menor do que P ou até encontrar a raiz. A complexidade dessa consulta é $O(QN)$, o que é proibitivo para a maior instância possível.

Busca binária: aplicação em grafos

Problema "My Ancestor"

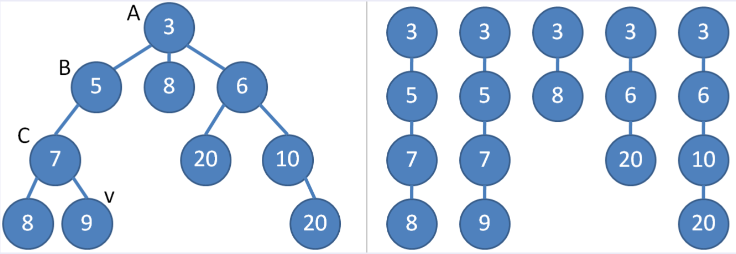


Problema “My Ancestor”

- Uma solução mais eficiente consiste primeiramente armazenar as Q consultas. Em seguida, realizamos uma única busca em profundidade na árvore (pré-ordem). Essa busca é adaptada para armazenar o caminho parcial da raiz até o nó atual em um vetor auxiliar . Esse vetor é ordenado devido à propriedade de heap de mínimo.
- Durante o percurso em profundidade, se visitamos um nó requisitado por alguma consulta, realizamos uma busca binária no caminho parcial (da raiz até nó atual) para encontrar o nó ancestral desejado.
- Como são Q consultas e cada consulta requer $O(\log N)$, temos uma complexidade $O(N + Q \log N)$.

Busca binária: aplicação em grafos

Problema “My Ancestor”



Referências

1. S. Halim e F. Halim. Competitive Programming 2, Second Edition Lulu (www.lulu.com), 2011. (IMECC – 005.1 H139c)
2. S. S. Skiena, M. A. Revilla. Programming Challenges: The Programming Contest Training Manual, Springer, 2003.
3. T.H. Cormen, C.E. Leiserson, R.L.Rivest e C. Stein. Introduction to Algorithms. 2nd Edition, McGraw-Hill, 2001. (no. chamada IMECC – 005.133 Ar64j 3.ed.)
4. U. Manber. Introduction to Algorithms: A Creative Approach. Addison-Wesley. 1989. (no. chamada IMECC – 005.133 Ec53t 2.ed.)