Resolução de problemas utilizando recursão

Programação de Computadores 1

Prof. Daniel Saad Nogueira Nunes

IFB – Instituto Federal de Brasília, Campus Taguatinga

Sumário

Introdução

- Agora que conhecemos o básico sobre recursividade, iremos nos aprofundar do seu uso para resolução de problemas.
- Iremos utilizar várias situações-problema para exemplificar a aplicação desta técnica de programação.

Sumário

Exponenciação

Problema

Dado $x \in \mathbb{R}$ e $n \in \mathbb{N}$, calcule x^n sem utilizar a função `pow`.

Exponenciação: recursão

- A exponenciação tem uma caracterização recursiva imediata.
- Se $n = 0$, $x^n = 1$.
- Senão, $x^n = x^{n-1} \cdot x$.

$$x^n = \begin{cases} 1, & n = 0 \\ x^{n-1} \cdot x, & n > 0 \end{cases}$$

Exponenciação: recursão

```
double expo(double x, int n) { return n == 0 ? 1 : expo(x, n - 1) * x; }
```

Exponenciação

- A solução recursiva faz $n + 1$ chamadas.
- Memória utilizada é proporcional ao valor de n.
- A solução iterativa padrão se mostra uma escolha melhor. Apesar de efetuar n multiplicações, não utiliza memória de pilha em excesso.

Exponenciação: iteração

```
double expo(double x, int n) {
    double res = 1;
    for (int i = 1; i <= n; i++) {
        res *= x;
    }
    return res;
}
```

Exponenciação

- Podemos fazer melhor ao definir a solução recursiva de um jeito mais esperto.
- Esta técnica é conhecida como **exponenciação rápida**. Pode ser utilizada para gerar uma matriz M^n rapidamente.

Exponenciação rápida

- Caso base: se $n = 0$, $x^n = 1$.
- Se $n > 0$ e par, temos que:

$$x^n = x^{\frac{n}{2}} \cdot x^{\frac{n}{2}} = (x^{\frac{n}{2}})^2$$

- Se $n > 0$ e ímpar, temos que:

$$x^n = x \cdot x^{\frac{n-1}{2}} \cdot x^{\frac{n-1}{2}} = x \cdot x^{\lfloor \frac{n}{2} \rfloor} \cdot x^{\lfloor \frac{n}{2} \rfloor} = x \cdot (x^{\lfloor \frac{n}{2} \rfloor})^2$$

Exponenciação rápida

```
double expo(double x, int n) {
    if (n == 0)
        return 1;
    else {
        double pot = expo(x, n / 2);
        if (n % 2 == 0) {
            return pot * pot;
        } else {
            return x * pot * pot;
        }
    }
}
```

Exponenciação rápida

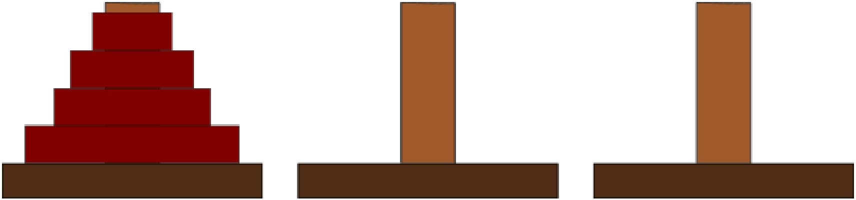
- A nova solução recursiva é muito mais interessante.
- A chamada recursiva reduz o valor do exponente pela metade.
- São feitas $1 + \lceil \log_2(n+1) \rceil$ chamadas.
- Complexidade **logarítmica** em vez de linear.

Sumário

Torre de Hanoi

- A torre de Hanoi é um quebra-cabeça que possui 3 estacas, rotuladas A, B e C e cinco discos, de tamanhos crescentes.
- Inicialmente os cinco discos estão empilhados, do menor para o maior, na primeira estaca, A.
- O objetivo é transferir todos os discos para a estaca C, podendo utilizar a estaca B durante o processo.
- **Restrição**: nunca um disco maior pode ficar sobre um menor.

Torre de Hanoi



Torre de Hanoi

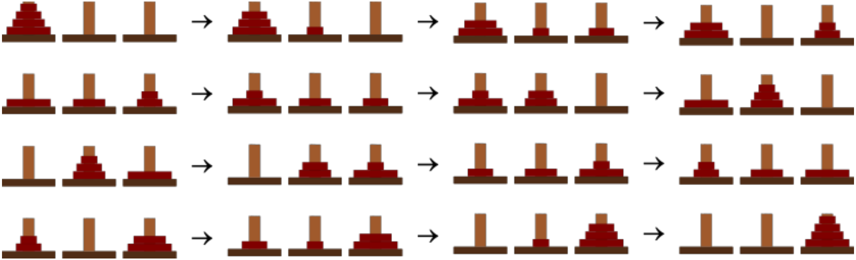


Figura: Solução do quebra-cabeça com 4 discos.

Torre de Hanoi

- Todo quebra-cabeça da torre de Hanoi com n discos pode ser resolvido com $2^n - 1$ movimentos.

Torre de Hanoi

Problema

Resolva um quebra-cabeça da Torre de Hanoi com n discos, informando as movimentações dos discos para cada uma das estacas com $2^n - 1$ movimentos.

Torre de Hanoi

- Caso base: se $n = 1$ resolver o problema é trivial, basta mover o disco da estaca origem para a estaca destino.
- Se soubermos resolver o quebra-cabeça com $n - 1$ discos (hipótese de indução), também conseguimos resolver o quebra-cabeça com n discos:
  - Transferimos os $n - 1$ primeiros discos para a estaca auxiliar utilizando a solução recursiva para $n - 1$.
  - Transferimos o disco n para a estaca destino.
  - Transferimos os $n - 1$ discos da estaca auxiliar para a estaca destino utilizando a solução recursiva para $n - 1$ discos.

Torre de Hanoi

```c
void hanoi(int n, char origem, char destino, char auxiliar) {
    if (n == 1) { // Caso base
        printf("Movendo o disco %d da estaca %c para a estaca %c.\n", n, origem, destino);
    } else {
        // Movemos os "n - 1" primeiros discos da estaca origem para a estaca
        // auxiliar utilizando a estaca de destino como auxiliar.
        hanoi(n - 1, origem, auxiliar, destino);
        // Movemos o disco "n" da estaca origem para a estaca destino
        printf("Movendo o disco %d da estaca %c para a estaca %c.\n", n, origem, destino);
        // Movemos os "n - 1" primeiros discos da estaca auxiliar para a estaca
        // destino utilizando a estaca origem como auxiliar.
        hanoi(n - 1, auxiliar, destino, origem);
    }
}

```

Sumário

Sistema linear

- A recursividade é extremamente útil quando queremos enumerar todas as possibilidades, subconjuntos ou permutações.
- Examinaremos alguns exemplos que utilizam a recursão desta forma.

Sistema linear

Problema

Sejam inteiros n e C e uma equação $x_0 + x_1 + \ldots + x_n = C$, informe todos os valores das variáveis que satisfazem a equação.

Sistema linear

Restrições

- $x_i \geq 0,\ 0 \leq i \leq n.$
- $C \geq 0.$

Sistema linear

- Se o inteiro n fosse conhecido previamente, poderíamos implementar a solução iterativamente.
- Por exemplo, se $n = 2$ bastaria fazer o seguinte:

```
void solution(int C) {
    int x0, x1, x2;
    for (x0 = 0; x0 <= C; x0++) {
        for (x1 = 0; x1 <= C - x0; x1++) {
            x2 = C - x0 - x1;
            printf("%d + %d + %d = %d", x0, x1, x2, C);
        }
    }
}
```

Sistema linear

- Como não sabemos o valor de n de antemão, a solução anterior não funcionaria.
- Utilizando recursividade, podemos enumerar todas as soluções.

Sistema linear: caracterização recursiva

- Seja o sistema

$$x_0 + x_1 + \ldots + x_{n-1} + x_n = C$$

- Se $n = 0$, a única variável deve assumir o valor de C para satisfazer o sistema, isto é, $x_0 = C$.
- Se $n > 0$, podemos estipular um valor válido para x_n e resolver, recursivamente o problema menor

$$x_0 + x_1 + \ldots + x_{n-1} = C - x_n$$

Sistema linear: solução recursiva

```c
void imprime_solucao(int x[], int n, int C) {
    for (int i = 0; i < n; i++) {
        printf("%d + ", x[i]);
    }
    printf("%d = %d\n", x[n], C);
}
```

Sistema linear: solução recursiva

```
void solucao(int *x, int n, int C, int n_original, int c_original) {
    if (n == 0) {
        x[0] = C;
        imprime_solucao(x, n_original, c_original);
    } else {
        for (int i = 0; i <= C; i++) {
            x[n] = i;
            solucao(x, n - 1, C - x[n], n_original, c_original);
        }
    }
}
```

Sistema linear: solução recursiva

```c
int main(void) {
    int n, C;
    scanf("%d %d", &n, &C);
    int *x = malloc(sizeof(int) * n + 1);
    solucao(x, n, C, n, C);
    free(x);
    return 0;
}
```

Sumário

Permutação

Problema

Dado uma string S de tamanho n, imprimir todas as permutações de S

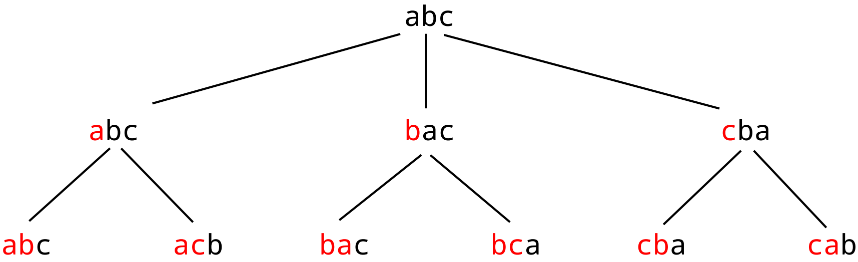
Permutação

- Por exemplo, se $S = abc$ então as permutações são: abc, bac, cba, acb, bca, cab.
- Total de permutações: $n!$.

Permutação

- A ideia aqui é a seguinte: para cara caractere $S[l]$, trocamos ele por um caractere $S[i]$, com $i \geq l$.
- Resolvemos o problema recursivamente para a string $S[l+1, n-1]$.
- O processo acaba quando chegamos na última posição.

Permutação



Permutação

```c
void solucao(char *str, int l, int r) {
    if (l == r) {
        printf("%s\n", str);
    } else {
        for (int i = l; i <= r; i++) {
            swap(str, i, l);
            solucao(str, l + 1, r);
            swap(str, i, l);
        }
    }
}
```

Permutação

```c
void swap(char *str, int i, int j) {
    char swap = str[i];
    str[i] = str[j];
    str[j] = swap;
}
```

Permutação

```
int main(void) {
    char str[30];
    scanf("%s", str);
    solucao(str, 0, strlen(str) - 1);
    return 0;
}
```

Sumário

Subset sum

Problema

Seja $V = (v_0, \ldots, v_{n-1})$ uma sequência de inteiros de tamanho n e C um inteiro. Verifique se é possível tomar um subconjunto $S \subseteq [0, n-1]$ tal que $\sum_{x \in S} V[x] = C$.

Subset sum

- Por exemplo, se $V = (30, 40, 10, 15, 10, 60, 54)$ e $C = 55$ o conjunto $S = 1, 3$ satisfaz a solução, pois $V[1] + V[3] = 55$.
- A resposta é sim.
- Se $V = (5, 6, 3, 5, 3, 6)$ e $C = 7$, não há subconjunto que satisfaça o problema.
- A resposta é não.

Subset sum

- A estratégia imediata é clara: gerar todos os subconjuntos S.
- Cada elemento $0 \leq i < n$ pode estar ou não estar em S.
- Testamos as duas possibilidades.
- Total de subconjuntos: 2^n.

Subset sum

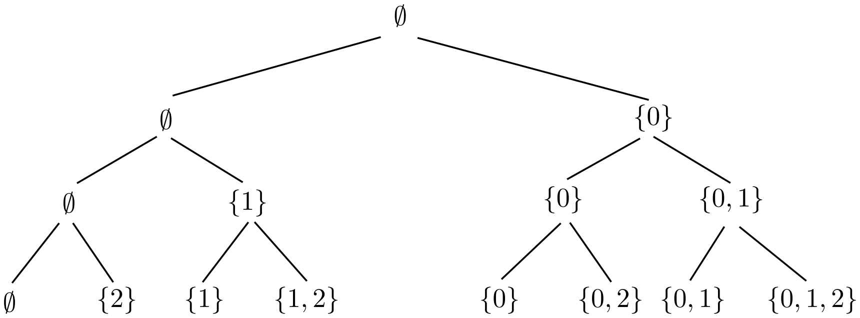


Figura: Subconjuntos de $\{0, 1, 2\}$

Subset sum

```c
int solucao(int *v, int i, int n, int X, int soma) {
    if (soma > X) {
        return 0;
    }
    if (i == n) {
        return soma == X;
    }
    return solucao(v, i + 1, n, X, soma) ||
           solucao(v, i + 1, n, X, soma + v[i]);
}
```

Subset sum

```
int main(void) {
    int n, X;
    scanf("%d %d", &n, &X);
    int *v = malloc(sizeof(int) * n);
    for (int i = 0; i < n; i++) {
        scanf("%d", &v[i]);
    }
    if (solucao(v, 0, n, X, 0)) {
        printf("Sim\n");
    } else {
        printf("Nao\n");
    }
    free(v);
    return 0;
}
```