Prolem Solving Paradigms

Força Bruta / Backtracking / Greedy

Profs. André, Salles

Departamento de Informática
Universidade Federal de Viçosa

INF333 - 2024/1

Introdução

- Computadores atuais são tão rápidos que alguns problemas podem ser resolvidos na força bruta (busca exaustiva, testar todas as alternativas)
- Contar os elementos de um conjunto, por exemplo. Em alguns casos é mais fácil gerar todos eles que encontrar uma fórmula por análise combinatória
- É claro, isso só será possível se o número de itens for suficientemente pequeno para ter tempo de gerar e contar todos eles

Introdução

Exemplo: UVa 725 - Division (link)

Dado um inteiro positivo N, encontrar e escrever todos os pares de inteiros de 5 dígitos que usam todos os dígitos de 0 a 9 (exatamente uma vez cada), de tal forma que o primeiro dividido pelo segundo seja igual a N.

Isto é, encontrar todos `abcde / fghij =` N, sendo cada letra um dígito diferente (podem começar com 0).

Ex.: Para $N = 62$, temos 79546 / 01283 = 62 e 94736 / 01528 = 62.

Introdução

Exemplo: UVa 725 - Division (link)

Dado um inteiro positivo N, encontrar e escrever todos os pares de inteiros de 5 dígitos que usam todos os dígitos de 0 a 9 (exatamente uma vez cada), de tal forma que o primeiro dividido pelo segundo seja igual a N.

Isto é, encontrar todos `abcde / fghij` $= N$, sendo cada letra um dígito diferente (podem começar com 0).

Ex.: Para $N = 62$, temos 79546 / 01283 = 62 e 94736 / 01528 = 62.

Solução 1

- Verificar todas as permutações dos dígitos em `abcdefghij`
- Quantas verificações? $10! = 3.628.800$

Introdução

Exemplo: UVa 725 - Division (link)

Dado um inteiro positivo N, encontrar e escrever todos os pares de inteiros de 5 dígitos que usam todos os dígitos de 0 a 9 (exatamente uma vez cada), de tal forma que o primeiro dividido pelo segundo seja igual a N.

Isto é, encontrar todos `abcde / fghij` $= N$, sendo cada letra um dígito diferente (podem começar com 0).

Ex.: Para $N = 62$, temos 79546 / 01283 = 62 e 94736 / 01528 = 62.

Solução 2

- Testar `fghij` = 01234 até 98765
- Para cada um, multiplicar por N e verificar se os dígitos são diferentes
- Quantas verificações? < 100.000
- Essa ideia de "pensar ao contrário" ajuda em vários problemas! Dica: anotar isso na "lista de técnicas" (que deve ser levada para a maratona)

Exemplo: UVa 11565 - Simple Equations (link)

Dados três inteiros A, B, C, com $1 \leq A, B, C \leq 10000$, encontrar três inteiros distintos x, y, z tal que $x + y + z = A$, $xyz = B$ e $x^2 + y^2 + z^2 = C$.

Ex.: Para $A = 6, B = 6, C = 14$, uma solução é $x = 1, y = 2, z = 3$.

Introdução

Exemplo: UVa 11565 - Simple Equations (link)

Dados três inteiros A, B, C, com $1 \leq A, B, C \leq 10000$, encontrar três inteiros distintos x, y, z tal que $x + y + z = A$, $xyz = B$ e $x^2 + y^2 + z^2 = C$.

Ex.: Para $A = 6, B = 6, C = 14$, uma solução é $x = 1, y = 2, z = 3$.

Solução

- Como $C \leq 10000$, temos que $x^2 \leq 10000$. O mesmo vale para y e z.
- Logo, $x, y, z \in [-100, 100]$
- Um loop triplo resultaria em $\approx 8M$ a serem testados, no pior caso
- Um loop duplo em x, y bastaria, calculando z com $z = A - x - y$

Introdução

- Computadores atuais tem clock na faixa de Gigahertz
- Então eles podem realizar bilhões de instruções por segundo
- Sabendo que qualquer cálculo interessante gasta 100 ou mais instruções, será possível enumerar ou contar um conjunto contendo milhões de elementos

- O que significa 1 milhão de elementos?
  - todas as possíveis permutações de 10 itens (são 3.628.800)
  - todas os possíveis subconjuntos de 20 itens (são 1.048.576)

- Para problemas maiores que esses, é necessário um esquema mais esperto, que verifique somente os elementos que realmente interessam

Backtracking

É um método para gerar/enumerar/percorrer sistematicamente todas as alternativas em um espaço de soluções.

É uma técnica geral que deve ser adaptada para cada aplicação.

- Aplicado quando for possível trabalhar com uma *solução parcial*
- O método constrói soluções candidatas incrementalmente e abandona uma solução parcial quando identifica que tal solução parcial não levará a uma solução do problema
- Em cada passo do processo de enumeração são tentadas todas as possíveis alternativas recursivamente
- É um método de tentativa e erro: tenta um caminho; se não der certo, volta e tenta outro, marcando os caminhos por onde passa

Backtracking

Algoritmo geral

- Dada uma solução parcial C com elementos candidatos $c_1, c_2, \ldots, c_k$
- Se é uma solução
  - contar, imprimir, comparar, etc
- Se não é, verificar se pode adicionar mais um elemento
  - Se puder, adicionar c_{k+1} em C e continuar recursivamente
  - Apagar o último c_{k+1} adicionado e tentar outro iterativamente

- Método correto: verifica todas as alternativas
- Método “eficiente”: não testa a mesma alternativa mais de uma vez
- mas... só deve ser usado se realmente é necessário avaliar todas as alternativas

Backtracking - exemplos de aplicações

- Quais são todos os subconjuntos de $\{2, 5, 8, 9, 13\}$?
- Em quantos destes a soma dos elementos é ≤ 17 ?
- Quais são todas as permutações de $[2, 5, 8, 9, 13]$?
- Em quantas destas os itens 8 e 9 não aparecem juntos ?
- Dado um labirinto, achar um caminho para sair dele
- Colocar 8 rainhas num tabuleiro de xadrez sem que se ataquem
- Resolver um quebra-cabeças com várias configurações e apenas uma (ou algumas) correta
- ...

Backtracking

- Algoritmo geral

```
// int c[] - solução parcial, de c[0] até c[k-1]
// int k - posição para inserir o próximo passo/candidato
// int n - tamanho máximo da solução

void backtrack(int c[], int k, int n)
{
    if ( ## terminou ##) {          // se chegou numa possível solução
        ## processa solucao ##;
        return;
    }

    for ( ## toda alternativa ## )          // para toda alternativa
        if ( ## alternativa viável ## ) { // se pode incluí-la
            c[k] = ## alternativa ##;     // inclui na solução parcial
            backtrack(c, k+1, n);         // gera o restante
        }
}
```

Backtracking - gerar todos os subconjuntos

```
void gerasub(bool c[], int k, int n)
{
    if (k == n) {          // se terminou de gerar um subconjunto
        imprimesub(c, n);
        return;
    }

    c[k] = true;           // subconjuntos com o elemento k
    gerasub(c, k+1, n);
    c[k] = false;          // subconjuntos sem o elemento k
    gerasub(c, k+1, n);
}

int main()
{
    bool c[1000];          // subconjunto parcial
    int n;                 // tamanho do conjunto

    cin >> n;
    gerasub(c, 0, n);    // subconjuntos de {0, 1, 2, 3, ..., n-1}
}
```

Backtracking - gerar todas as permutações

```
void geraperm(int c[], int k, int n)
{
    if (k == n) {          // se terminou de gerar a permutação
        imprimeperm(c, n);
        return;
    }

    for(int i=0;i<n;i++)           // para todo elemento i
        if (!pertence(c,k,i)) {    // se i não está na permutação parcial
            c[k] = i;              // inclui o elemento na permutação
            geraperm(c, k+1, n);   // gera o restante da permutação
        }
}

int main()
{
    int c[1000];           // permutação parcial
    int n;                 // tamanho do conjunto

    cin >> n;
    geraperm(c, 0, n);     // permutações de {0, 1, 2, 3, ..., n-1}
}
```

Backtracking - cortes

- É importante **cortar** a busca tão logo se descubra que a solução parcial não levará a uma solução aceitável do problema

```
void backtrack(int c[], int k, int n)
{
    if ( ## não há como continuar ##)     // corte
        return;

    if ( ## terminou ##) {          // se chegou numa possível solução
        ## processa solucao ##;
        return;
    }

    for ( ## toda alternativa ## )          // para toda alternativa
        if ( ## alternativa viável ## ) { // se pode incluí-la
            c[k] = ## alternativa ##;     // inclui na solução parcial
            backtrack(c, k+1, n);         // gera o restante
        }
}
```

Backtracking

- Conceitualmente, é como uma árvore de busca
- As soluções candidatas são nós da árvore
- Cada nó é pai das soluções parciais com um passo a mais
- As folhas são soluções que não podem mais crescer

Backtracking - exemplo

Exemplo: UVa 750 - 8 Queens Chess Problem (link)

No tabuleiro de xadrez (8x8) é possível colocar 8 rainhas sem que nenhuma ataque outra.

Determine *todas* as possíveis configurações, dada a posição de uma das rainhas (ou seja, certa coordenada (a, b) deve conter uma rainha).

Backtracking - exemplo

Exemplo: UVa 750 - 8 Queens Chess Problem (link)

No tabuleiro de xadrez (8x8) é possível colocar 8 rainhas sem que nenhuma ataque outra.

Determine *todas* as possíveis configurações, dada a posição de uma das rainhas (ou seja, certa coordenada (a, b) deve conter uma rainha).

Solução 1

- Testar todas as configurações por força bruta
- Quantas verificações? $\binom{64}{8} = 4.426.165.368$

Backtracking - exemplo

Exemplo: UVa 750 - 8 Queens Chess Problem (link)

No tabuleiro de xadrez (8x8) é possível colocar 8 rainhas sem que nenhuma ataque outra.

Determine *todas* as possíveis configurações, dada a posição de uma das rainhas (ou seja, certa coordenada (a, b) deve conter uma rainha).

Solução 2

- Cada rainha deve estar em uma linha diferente
- Então basta escolher uma coluna para cada linha
- Quantas verificações? $8^8 = 16.777.216$

Backtracking - exemplo

Exemplo: UVa 750 - 8 Queens Chess Problem (link)

No tabuleiro de xadrez (8x8) é possível colocar 8 rainhas sem que nenhuma ataque outra.

Determine *todas* as possíveis configurações, dada a posição de uma das rainhas (ou seja, certa coordenada (a, b) deve conter uma rainha).

Solução 3

- Cada rainha deve estar em uma linha e uma coluna diferente
- Então basta escolher uma coluna diferente para cada linha
- Permutações de $[1, 2, 3, 4, 5, 6, 7, 8]$
- Quantas verificações? $8! = 40.320$

Backtracking - exemplo

Exemplo: UVa 750 - 8 Queens Chess Problem (link)

No tabuleiro de xadrez (8x8) é possível colocar 8 rainhas sem que nenhuma ataque outra.

Determine *todas* as possíveis configurações, dada a posição de uma das rainhas (ou seja, certa coordenada (a, b) deve conter uma rainha).

Solução 3b

- Além disso não podem estar na mesma diagonal
- Cortar a busca se a nova rainha está na mesma diagonal das anteriores
  - (i, j) e (k, l) estão na mesma diagonal se $abs(i - k) == abs(j - l)$

Backtracking - dicas

- A grande aposta em usar Força Bruta para resolver um problema é se passa no *Time Limit*
- Se TL = 1 min e seu programa executa em 1 min 5 s, você pode tentar melhorar a parte crítica do código, em vez de começar do zero um algoritmo mais rápido porém não trivial
- Um desafio é o fato de não sabermos exatamente o tempo na máquina de testes. De qualquer forma, faça testes com entradas grandes.

Backtracking - dica 1

Gerar x Filtrar

- Programas que geram muitas soluções candidatas e então escolhe as corretas (ou remove as incorretas) são chamados “filtros”
- Programas que geram exatamente as respostas corretas sem começos falsos são chamados “geradores”
- Geralmente programas “filtros” são mais fáceis de codificar, porém mais lentos
- Faça os cálculos para ver se um programa “filtro” dá conta ou se precisa escrever um “gerador”

Backtracking - dica 2

Cortar espaço de busca inviável mais cedo

- Verificar se uma solução parcial pode levar a uma solução completa
- Se não puder, cortar (retornar e tentar outra)

Backtracking - dica 3

Usar simetrias

- Alguns problemas possuem simetrias
- Usar esse fato para reduzir o tempo de execução

Backtracking - dica 4

Pré-processamento

- Pode ser útil gerar tabelas ou outras estruturas de dados que permitam buscar o resultado de forma mais rápida
- Antes de iniciar propriamente o programa, gerar todos (ou boa parte) desses dados
- Uso de memória/espaço em vez de tempo

Backtracking - dica 5

Tentar resolver de “trás pra frente”

- Curiosamente, alguns problemas são melhor resolvidos pensando ao contrário, ou começando do fim para o início
- Pode ser útil processar os dados em outra ordem que não a óbvia
- Ex.: UVa 10360 - Rat Attack (link)

Backtracking - dica 6

Otimizar o código

- Entender como cálculos são feitos em hardware ajuda a programar melhor
- Isso inclui entrada/saída, comportamento de cache, entre outros

1. Usar printf/scanf em vez de cin/cout
2. Usar quicksort em vez de merge/heapsort (usar `sort` da STL!)
3. Acessar matriz linha por linha em vez de coluna por coluna
4. Manipulação de bits é mais rápida que usar array de booleanos
5. Usar estruturas/tipos de nível mais baixo:
   ~~usar array em vez de vector~~ (melhor: usar vector como array, sem realocação), int em vez de long long (às vezes)
6. Declarar variáveis compostas gigantes uma única vez, com escopo global
7. Alocar memória uma única vez (com os limites fornecidos no enunciado) em vez de realocar dinamicamente par cada caso de teste

Backtracking - dica 7

Usar estruturas de dados e algoritmos melhores

- Sério, isso pode ser melhor que todas as outras 6 dicas!

Backtracking - dica 8

:-(

- Se tudo isso falhar, abandonar a Força Bruta...

Gerar todas as permutações - outra forma

Existe a função `next_permutation` na biblioteca `algorithm`!

```
int a[] = {1,2,3,4,5,6,7,8};

do {
   ...
} while(next_permutation(a,a+8));
```

Gerar todos os subconjuntos - outra forma

Usar a representação binária dos números $0 \ldots 2^n - 1$

```
for(int i=0; i<(1<<n); i++) {   \\para cada subconjunto
  for(int j=0; j<n; j++) {        \\para cada elemento
    if (i & (1 << j)) {             \\testa se bit j é true em i

      ... /* elemento j pertence ao subconjunto i */

    }
  }
}
```

Link: Tutorial do topcoder sobre bits
Link: Tutorial do geeksforgeeks
Link: Livro Hacker's Delight

Meet in the Middle

- Dividir o espaço de busca em duas partes de tamanhos similares
- Fazer busca exaustiva em cada parte e combinar os resultados

- Usada se os resultados das buscas podem ser combinados de forma eficiente; nesse caso, 2 buscas gastam menos que 1 maior
- Pode reduzir de 2^n para $2^{(n/2)}$
- Exemplo de aplicação: (link)

Greedy

- Outra técnica de projeto de algoritmos: métodos Gulosos
- Faz escolhas locais ótimas em cada passo na esperança de encontrar o ótimo global
  - Em alguns casos funciona: com vantagem de código pequeno e eficiente
  - Mas em *muitos* casos não funciona
- Requisitos
  - Sub-estrutura ótima:
    soluções ótimas contém soluções ótimas dos sub-problemas
  - Ter uma propriedade “gulosa”: (obs. geralmente difícil de provar)
    Escolhe o que parece melhor no momento, depois resolve os sub-problemas restantes, e ainda assim chega na solução ótima.
    Nunca é preciso reconsiderar uma escolha anterior.

Greedy

Problema: Coin change

Suponha um grande número de moedas de 50, 25, 10, 5 e 1.
Devolver troco com o menor número possível de moedas.

Solução greedy

Em cada passo usar o maior valor de moeda possível

Exemplo

Devolver 42.
42 - 25 = 17 - 10 = 7 - 5 = 2 - 1 = 1 - 1 = 0
5 moedas, $\{25, 10, 5, 1, 1\}$, solução ótima

Greedy

O método tem os dois ingredientes de um greedy de sucesso:

1. Sub-estrutura ótima
   - Solução ótima para 42: $\{25, 10, 5, 1, 1\} \rightarrow 5$ moedas
   - Esta solução contém as soluções ótimas dos subproblemas
   - Solução ótima para 17: $\{10, 5, 1, 1\} \rightarrow 4$ moedas
   - Solução ótima para 7: $\{5, 1, 1\} \rightarrow 3$ moedas
2. Propriedade greedy
   - Dada uma quantidade V, subtraímos *greedly* o maior valor de moeda
   - É possível provar que outras estratégias não garantem o ótimo

Mas...

- ele não funciona para qualquer conjunto de moedas
- por exemplo, devolver 6 com moedas $\{4, 3, 1\}$
- ele utiliza 3 moedas: $\{4, 1, 1\}$
- a solução ótima usa apenas 2 moedas: $\{3, 3\}$

Greedy

Exemplos de métodos greedy clássicos

- Kruskal para árvore geradora mínima
- Prim para árvore geradora mínima
- Dijkstra para caminho mínimo
- Código de Huffman

Obs.: os três primeiros serão vistos no assunto Grafos

Greedy

Exemplo: UVa 410 - Station Balance (link)

- Certo laboratório tem uma centrífuga com C câmaras
- Cada câmara pode conter 0, 1 ou 2 amostras
- Você deve distribuir S amostras de forma a minimizar o desbalanceamento: $IMBALANCE = \sum_{i=1}^{C} |CM_i - AM|$
  - CM_i: massa na câmara i (soma das massas das amostras nela)
  - AM: massa média geral (total das amostras dividido por C)
- Para a entrada $C = 2, S = 3$, massas $= \{6, 3, 8\}$
  - Solução: $C_1 = \{6, 3\}, C_2 = \{8\}, IMBALANCE = 1.00000$
- Para a entrada $C = 3, S = 5$, massas $= \{51, 19, 27, 14, 33\}$
  - $C_1 = \{51\}, C_2 = \{19, 27\}, C_3 = \{14, 33\}, IMBALANCE = 6.00000$

Greedy

Exemplo: UVa 10020 - Minimal Coverage (link)

- Há alguns intervalos de inteiros e um valor M
- Desejamos o mínimo de intervalos para cobrir [0,M]
- Exemplo: $M = 10$, $[0, 3]$,$[0, 1]$,$[3, 5]$,$[3, 10]$,$[2, 4]$,$[2, 3]$
  - Solução: $[0, 3]$, $[3, 10]$
- Ideia: ordenar com base no tamanho do intervalo.
  - $[3, 10]$,$[0, 3]$,$[2, 4]$,$[3, 5]$, $[0, 1]$,$[2, 3]$
  - $[3, 10]$ será escolhido primeiro ; $[0, 3]$ depois
- E se o segundo maior fosse $[3, 9]$?
- E se o maior fosse $[-100, -1]$?
- E o caso $[2, 9]$,$[1, 5]$,$[5, 10]$?
- Nem há garantias de que o maior intervalo estará na resposta!

Greedy

Exemplo: UVa 10020 - Minimal Coverage (link)

- Exemplo: $M = 10$, $[0, 3]$,$[0, 1]$,$[3, 5]$,$[3, 10]$,$[2, 4]$,$[2, 3]$
  - Solução: $[0, 3]$, $[3, 10]$
- Ponto chave 1: ordenar com base no início do intervalo (isso frequentemente é útil em gulosos)
  - $[0, 1]$,$[0, 3]$,$[2, 3]$,$[2, 4]$,$[3, 5]$,$[3, 10]$
- Ponto chave 2: precisamos cobrir o início de $[0, 10]$ e intervalo que cobre e termina depois nunca é pior
- Manter esquerda L do intervalo formado
- Processar intervalos I
  - Se $I.left > L \rightarrow$ não é possível cobrir
  - Senão, pegar o intervalo que termina mais à direita e com $I.left \leq L$
  - Atualizar L com a direita do melhor intervalo
  - Se L ficar $\geq M \rightarrow$ cobriu o intervalo

Greedy - comentários

- Usar soluções greedy em campeonatos de programação é arriscado
- Normalmente não recebem TLE, mas tendem a receber WA
- Provar que certo problema tem sub-estrutura ótima e propriedade greedy leva tempo (coisa escassa em campeonatos...)
- Técnica: olhar para o tamanho da entrada
  - Se for pequena o bastante para Força Bruta ou Programação Dinâmica, usar isso! Elas levam à resposta correta
  - Usar greedy somente se o tamanho da entrada for muito grande para o melhor FB ou DP que conseguir pensar
- Problema: quem escreve os problemas conhece essa técnica...
  - hoje em dia o tamanho é escolhido de tal forma que o competidor não adivinhe de cara se deve usar greedy ou não