Aula 05 - Bizus e Guloso

Kenji Yamane

Março de 2020

Tópicos da aula

EOF em C++

Na aula passada foi mostrado como fazer EOF com scanf. Aproveita-se para adicionar que caso você queira simular um "fim de arquivo" para verificar se o seu programa realmente para, pode-se apertar ctrl D no linux, e se for no windows basta apertar ctrl Z. Agora, como que se faz a mesma coisa com cin? É até mais simples, como a maioria das coisas em C++ são. Enquanto scanf retorna -1 ao não conseguir mais ler algo, cin retorna NULL (ponteiro que aponta para nada), que efetivamente funciona como uma booleana de falso. Portanto, em código (o endl é tipo um \n):

Código 1

```
while (cin » N)
 cout « N « endl;
```

Memset

Da biblioteca string.h, memset é uma função bem simples mas que é útil também, por ser prática. Caso você queira zerar todos os valores de uma matriz memo, por exemplo:

Código 2

```
memset(memo, 0, sizeof(memo));
```

De fato já é possível fazer isso com listas inicializadoras de C com menos código até, porém memset pode servir para inicializar todas as casas para -1 caso você queira:

Código 3

```
memset(memo, -1, sizeof(memo));
```

Isso vai ser bem útil quando visitarmos programação dinâmica posteriormente.

Obs: usem memset somente com 0 e -1 (estranho eu sei).

Estruturas como argumento de funções

Muitos devem lembrar de variáveis de escopo das aulas de C. Quando você passa uma variável para uma função, essa variável não é efetivamente passada, mas sim é criada uma cópia dela. A mesma coisa acontece caso você queira passar uma estrutura da STL. Porém, é possível passar a estrutura, sem ser somente uma cópia, para caso você queira fazer alterações que sejam observadas na main também:

Código 4

```
int func(int aux, vector<pair<int, int> > v); //passando cópia
int func(int aux, vector<pair<int, int> > &v); //passando o vetor
```

Sort - inversão

Lembram do sort? Uma das razões dele ser tão bizu é que ele ordena em O(NlogN), o mais rápido que tem para um algoritmo de ordenação em que o espaço amostral é genérico. Por padrão ele deixa na ordem crescente, mas é possível ordenar em ordem decrescente de uma forma parecida com quando se inverte a priority_queue:

Código 5

```
sort(v, v + N, greater<int>());
```

Sort - em outras coisas

Vocês podem ordenar outras estruturas mais complicadas também, basta dar ao sort a forma de comparação que você quer basear sua ordenação. Por exemplo se você quer ordenar um vetor de pairs em ordem crescente de primeiro do par, primeiro crie sua função comparativa:

Código 6

```
bool comparePairs(pair<int, int> a, pair<int, int> b)
 return (a.first < b.first);
```

E então insira no sort, (supondo v ser um vector de pairs):

Código 7

```
sort(v.begin(), v.end(), comparePairs);
```

Last touches

O seguinte código é uma feature que tem nas últimas versões de C++ (talvez seu compilador não tenha), é de varredura e funciona com deque, vector, set, map...

Código 8

```
for (auto c : mapa)
```

Escrever and e or ao invés de && e ||.

Código 9

```
if (aux1 == 8 and aux2 == 9 or aux3 < 10)
```

E, pode-se alterar o tipo de uma variável, se chama cast:

Código 10

```
double var = 9.8;
printf(% d\n, (int)var);
```

Assunto do dia

Agora sim falemos do foco da aula de hoje! O segundo paradigma do nosso planejamento, guloso.

- Força Bruta
- **Guloso**
- Dividir para Conquistar
- Programação Dinâmica

Como funciona o guloso

Considere uma resposta ótima outra forma de dizer resposta certa. Um algoritmo guloso é aquele que sempre escolhe localmente a opção ótima, com a esperança de no final alcançar a resposta ótima global. Por exemplo, considere o seguinte problema: dado um conjunto de n moedas cada uma com um valor diferente, e um determinado valor V, qual o número mínimo de moedas para que os valores delas somem V? Por exemplo, se $n = 4$ e moedas $= \{25, 10, 5, 1\}$ e $V = 22$, podemos expressar V como $\{1, 1, 5, 5, 10\}$ ou como $\{5, 5, 5, 5, 1, 1\}$, porém é requisitado o menor número possível de moedas! Perceba que é possível tentar algum algoritmo de força bruta, porém como normalmente acontece ele decerto seria muito lento (mais que $O(2^N)$).

Problema da moeda

Um algoritmo guloso seria sempre pegar a maior moeda que não ultrapasse o valor V. O problema localmente é escolher alguma moeda, e a escolha localmente ótima seria escolher a maior moeda possível com a esperança de, no final, ter o conjunto com o menor número possivel de moedas. Faz sentido, não? No caso de moedas = {25, 10, 5, 1} e V = 42, esse algoritmo pegaria primeiro 25, e faltaria 42 - 25 = 17 para completar V. Depois, 10, faltando então 17 - 10 = 7, depois 5, 1, e finalmente 1. O conjunto seria {25, 10, 5, 1, 1}, e a resposta 6, que de fato está certa. O código correspondente encontra-se no próximo slide.

Código 11


```
...
  int n, V, coins[10000], ans = 0, i;
  cin >> n;
  for (i = 0; i < n; i++) cin >> coins[i];
  cin >> V;

  sort(coins, coins + n, greater<int>());
  i = 0;
  while (V > 0)
    if (coins[i] > V) i++;
    else              V -= coins[i], ans++;
  cout << ans << endl;
  return 0;
}
```

Problema da moeda

Bem rápido! Algo em torno de O(NlogN). Porém **receberia wrong answer**. A resposta para esse problema não é um algoritmo guloso. Considere um outro caso, coins = {4, 3, 1} e V = 6. Por guloso, a solução seria {4, 1, 1}, mas outra possibilidade é {3, 3}, menor ainda. O que quer dizer que procurar pelo ótimo local não irá te levar ao ótimo global, nesse caso! Porém há problemas em que isso vale. Diz-se que eles tem a **propriedade gulosa**, algo que vocês verão que é bem difícil de se confirmar se de fato tem. Pode-se concluir assim que esse paradigma tende a levar *wrong answer*, porém evita bastante *time limit exceeded*. A solução de fato é por programação dinâmica, o último paradigma a ser visto.

Comprando gasolina

Um outro problema (dessa vez em que o guloso funciona): suponha que existam n distribuidoras de gasolina, cada uma vendendo a um preço p_i ao litro e com e_i litros no estoque. Se você precisa de V litros de gasolina, qual a menor quantia possível que você gastaria? Considerar o problema local, que seria a escolha da distribuidora, e escolher a opção ótima que é a que vender mais barato funciona nesse caso! O problema possui a propriedade gulosa. O código está amostrado a seguir.

Código 12


```
    ...
    int n, V, ans = 0;
    pair<int, int> gas[10000]; //pi, ei
    cin >> n;
    for (int i = 0; i < n; i++)
      cin >> gas[i].first >> gas[i].second;
    cin >> V;
    sort(gas, gas + n, comparePair);
    //comparePair: a.first < b.first
    for (int i = 0; V > 0; i++)
      ans += min(V, gas[i].second)*gas[i].first,
      V -= min(V, gas[i].second);
    cout << ans << endl;
    return 0;
}
```

Cobrindo o intervalo

Normalmente nos códigos de algoritmos gulosos aparece a ordenação dos dados, para poder ordená-los de acordo com a busca local pelo ótimo que você quer fazer. Em outro problema, dados N segmentos de linha (no eixo x), e suas abscissas de início e fim $[ini_i, fim_i]$, qual a quantidade mínima de segmentos que você precisa para cobrir um segmento [0, S]? O problema localmente é qual segmento você escolhe, e gulosamente se escolhe sempre o segmento com menor abscissa de início. Se as abscissas de início forem iguais, escolhe-se gulosamente a que tiver maior abscissa de fim. Porém, se a absicssa de fim for menor do que eu cobri até agora, eu a ignoro, e se a abscissa de início for maior do que eu cobri até agora, é impossível cobrir. É outro problema com a propriedade gulosa.

Código 13


```
#include <bits/stdc++.h>
#define pii pair<int, int>
using namespace std;

int N, S, reach, ans;
//reach eh ate onde foi coberto, comeca no 0
pii seg[10000]; //ini, fim

bool comparePair(pii a, pii b){
        if (a.first == b.first)
                return a.second > b.second;
        else
                return a.first < b.first;
}
```

Código 13


```
int main(){
  cin >> N;
  for (int i = 0; i < N; i++)
    cin >> seg[i].first >> seg[i].second;
  cin >> S;
  sort(seg, seg + N, comparePair);
  for (int i = 0; i < N; i++)
    if (seg[i].first <= reach){
      if (seg[i].second > reach)
        reach = seg[i].second, ans++;
    }
    else cout << "impossible to cover!" << endl
  cout << ans << endl;
  return 0;
}
```

Apagando e Ganhando

Comentando sobre o problema do contest de vocês, apagando e ganhando, alguns devem ter sido tentados por uma ideia gulosa que na realidade dava wrong answer. Relembrando o problema, dado um determinado número, e uma quantidade de dígitos a serem deletados, determine o maior número que pode ser gerado nessa brincadeira. Como alguns perceberam, sempre buscar deletar gulosamente os menores dígitos não funciona. Porém um algoritmo accepted envolve um príncipio guloso: sempre deletar de modo a trazer os maiores dígitos para esquerda.

Problemas de guloso

- Maximum Sum (II)
- Converter Quilômetros para Milhas
- Guerra
- Produção em Ecaterimburgo
- Comércio de Vinhos na Gergóvia