INSTITUTO FED. DE EDUCAÇÃO, CIÊNC. E TEC. DE PERNAMBUCO
CURSO: TEC. EM ANÁLISE E DESENVOLVIMENTO DE SISTEMAS
DISCIPLINA: ALGORITMOS E ESTRUTURAS DE DADOS
PROFESSOR: RAMIDE DANTAS
ASSUNTO: BACKTRACKING E DIVIDIR P/ CONQUISTAR

Aluno (a):			
Matrícula:		**Data:**	

Prática 11

Parte 0: Preparação

Passo 1: Crie um novo projeto chamado **Pratica11**.

Passo 2: Adicione os arquivos que acompanham a prática 11 ao projeto.

> **subsetsum.cpp**: resolve o problema do Subconjunto com Soma K. Contém um função main e implementações usando Força bruta e Backtracking.
>
> **subseqmax.cpp**: resolve o problema da Subsequência com Soma Máxima. Contém uma função main e implementações usando Dividir p/Conquistar.
>
> **util.h** e **util.cpp:** funções auxiliares.

Passo 3: Compile e rode o código para verificar se não há erros.

> Nesse ponto deve apenas compilar e rodar mas não produzir resultados corretos; algumas funções não estão implementadas.

Passo 4: Estude o código para se familiarizar.

> Veja o material de aula se necessário. Faça comentários nos trechos mais complicados. Refatore o código se achar que vai ajudar seu trabalho. Teste novamente antes de fazer modificações mais profundas.

Parte 1: Problema Subconjunto com Soma K

Passo 1: Estude o código **subsetsum.cpp**.

> Nesse código o problema é resolvido de duas formas: `subsetsumBF()` e `subsetsumBT()`. A primeira usa força bruta (Brute Force, BF), varrendo todas as combinações de subconjuntos para determinar se alguma tem valor igual a K (variável `value`). A segunda usa Backtracking (BT), gerando progressivamente os subconjuntos e testando se resultam na soma desejada. Na versão atual, ambas as funções tem complexidade média de $O(2^N)$. Sua tarefa é entender o código e implementar melhorias descritas nos próximos passos na versão 2 da função com Backtracking.

Passo 2: Adicione *pruning* ao código de `__subsetSumBTv2()`:

> A primeira melhoria é eliminar caminhos não promissores. Para isso, adicione uma verificação para testar se o subconjunto atual já possui uma soma maior que o valor desejado, o que indica que adicionar qualquer outro número a ele é inútil (obs.: isso só vale porque o array só possui valores positivos). Nesse acaso, aborte a busca nesse momento, para que o algoritmo continue em outro caminho.

Passo 3: Tratando um "pior caso" em `subsetSumBTv2()`:

Com a melhoria anterior o tempo de execução médio é melhorado bastante. Porém, quando é dado um número muito grande e para qual não há subconjunto cuja soma resulte nele, o tempo de busca acaba sendo próximo da força bruta, pois o algoritmo com Backtracking acaba testando todas as combinações (veja a variável `count`).

Adicione um trecho de código que verifica, antes de iniciar o backtracking, se o número fornecido já é maior que a soma de todos os outros números do array. Nesse caso, aborte com falha imediatamente.

Passo 4: Compile e rode a aplicação.

Faça testes modificando o tamanho do array, começando com valores pequenos enquanto testa o código. Aumente os valores para ver o impacto no tempo e número de subconjuntos testados (`count`). Valores acima de 30 podem demorar muito para rodar devido a natureza exponencial dos algoritmos.

Passo 5: (Desafio) Ordenando para acelerar em `subsetSumBTv2()`:

Considerando que a ordem dos elementos não é relevante, é possível reordenar o array do maior para o menor antes de rodar o algoritmo BT. Dessa forma o algoritmo vai tender a encontrar subconjuntos com menos elementos que resultem na soma K (pois ele tenta primeiro combinar números grandes para só por último combinar números pequenos). Isso implica que o algoritmo teve que descer menos na árvore de busca. O preço dessa alteração é o custo de ordenar o array antes.

Parte 2: Problema da Subsequência com Soma Máxima

Passo 1: Estude o código em **subseqmax.cpp**:

A função `subseqMaxBF()` resolve o problema de forma ingênua com complexidade $O(N^2)$. A função `subseqMaxDC()` é o ponto de entrada pra a solução usando Dividir p/ Conquistar, que roda em $O(N \log N)$. Essa função apenas chama `__subseqMaxDC()`, que é a função que realmente calcula a sequência de soma máxima de forma recursiva. Essa função usa `subseqMaxMiddle()`, que acha a sequência de soma máxima que passa pelo meio do array (`middle`), onde meio é um ponto entre inicio (`start`) e final (`finish`). O valor da maior soma é retornada pelas funções, e o intervalo da sequência é salva em `ini` e `end`.

Passo 2: Implemente a função `__subseqMaxDC()` em **subseqmax.cpp**:

Codifique a função `__subseqMaxDC`() de acordo com os comentários no código.

Passo 3: Compile e rode a aplicação.

Faça testes modificando o tamanho do array, começando com valores pequenos enquanto testa o código. Aumente os valores para ver o impacto no tempo e número de somas (`count`). Como os algoritmos são polinomiais é possível testar com valores grandes (acima de 100).