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: ALG. GULOSOS E PROGRAMAÇÃO DINÂMICA

Aluno (a):			
Matrícula:		**Data:**	

Prática 12

Parte 0: Preparação

Passo 1: Crie um novo projeto chamado **Pratica12**.

> Essa prática é uma extensão da prática 11, porém recomenda-se criar projetos separados.

Passo 2: Adicione os arquivos que acompanham a prática 12 ao projeto.

> **subsetsum.cpp**: (idêntico a prática 11) resolve o problema do Subconjunto com Soma K. Contém uma função main e implementações usando Força bruta e Backtracking.
>
> **subseqmax.cpp**: (modificado com relação a prática 11) resolve o problema da Subsequência com Soma Máxima. Contém uma função main e implementações usando Dividir p/Conquistar.
>
> **util.h** e **util.cpp:** (idênticos a prática 11) 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: Subconjunto com Soma K usando Algoritmos Gulosos

Passo 1: Crie uma a função `subsetsumGreedy()` em **subsetsum.cpp**.

> Na prática anterior resolvemos o problema usando força bruta e backtracking. Crie agora uma função baseada no princípio guloso: a cada passo escolha o que for imediatamente melhor; repita até achar uma solução ou desistir. No caso do problema de Soma K é possível aplicar a ideia do problema do Troco: escolha a cada passo o elemento do array com valor mais próximo de K (mas ainda menor que K); subtraia esse valor de K e repita o processo até que K seja 0 (caso de sucesso), K < 0 (falha) ou não haja mais elementos e K > 0 (falha). Veja que esse algoritmo é uma heurística e portanto não garante achar a solução do problema.

Passo 2: 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. Veja quantas vezes a heurística gulosa foi capaz de resolver o problema e compare o tempo dela com os algoritmos anteriores.

Passo 3: (Desafio) Melhore a heurística gulosa

Tente melhorar a heurística buscando outros critérios para selecionar os elementos a cada passo.

Parte 2: Subsequência de Soma Máxima (SSM) usando Programação Dinâmica (PD)

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

Foi adicionada a função `subseqMaxRec()` que resolve o problema da subsequência de soma máxima (SSM) de forma recursiva com complexidade $O(N^2)$, idêntica a da solução ingênua. A função `seqMax()` retorna a soma da SSM terminada exatamente em `pos` (a SSM pode ser composta apenas por `array[pos]`); a variável `ini` é preenchida com a posição de início da SSM terminada em `pos`. Em `subseqMaxRec()`, as posições do array são varridas para encontrar a SSM global. A posição da SSM global delimita o final da sequência; o valor de `ini` associado a melhor SSM marca o início da sequência máxima.

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

Codifique a função `subseqMaxMemo`() e `seqMaxMemo()` com base em `subseqMaxRec()` e `seqMax()` de forma a memorizar os valores de Soma e Início (`ini`) em arrays próprios para que não seja preciso recomputá-los repetidamente.

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).

Passo 4: (Desafio) Implemente a função `subseqMaxPD()` em **subseqmax.cpp:**

Percebendo a ordem com que `seqMax()` é invocada dentro de `subseqMaxRec()` é possível construir uma implementação *botton-up* com complexidade linear, $O(N)$, onde as SSMs locais são calculadas apenas uma vez, começando da posição 0 até o final do array. A medida que é feito o cálculo das SSMs locais, é verificado se é a maior SSM até o momento, de forma que ao terminar de varrer o array já temos a SSM global. Esse algoritmo é conhecido como algoritmo de Kadane.