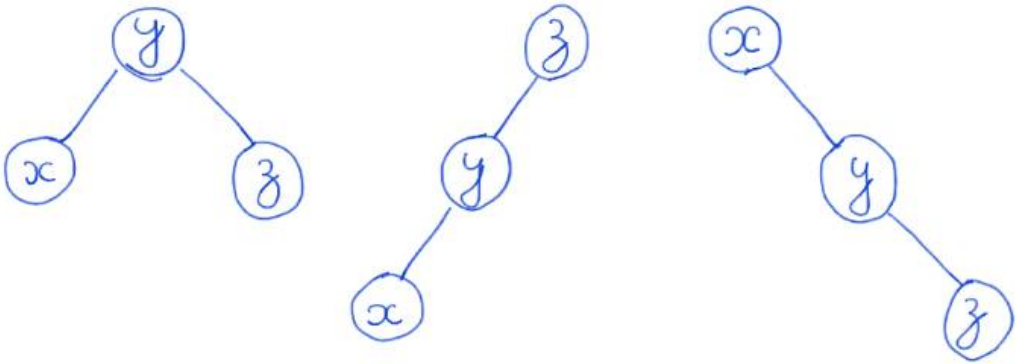
PAA - Aula 19
Problema da Árvore Binária de Busca Ótima

Dado um conjunto de itens, qualquer árvore binária de busca

- deve respeitar a propriedade de que os itens da subárvore esquerda
    - são menores que a raiz, que, por sua vez
        - é menor que os itens da subárvore direita.
- No entanto, existem inúmeras árvores binárias de busca válidas
    - para um mesmo conjunto de itens.
- Por exemplo, considere itens x < y < z.
    - Quais as árvores binárias de busca válidas para estes três itens?



- Assim surge a questão: qual a melhor árvore para busca?

Note que, o tempo de acesso a um item depende da profundidade deste na árvore,

- sendo profundidade o número de nós do caminho da raiz até o item.

Assim, pensando numa árvore de propósito geral e numa análise de pior caso,

- a melhor árvore é aquela que tem a menor altura, lembrando que altura
    - é o comprimento do caminho mais longo da raiz até um item.
- Portanto, uma árvore binária de busca balanceada, como
    - uma árvore AVL ou Rubro-Negra, é assintoticamente ótima,
        - por ter altura O(log n), sendo ‘n’ o número de nós na árvore.

Sendo este o caso, o que queremos dizer com Árvores Binárias de Busca Ótimas?

- Este problema surge em contextos em que conhecemos
    - a frequência/probabilidade com que os itens são acessados.
- Por exemplo, porque eles são palavras de uma linguagem
    - num software de verificação de erros gramaticais.
- Nestes casos vamos buscar algo melhor do que árvores de propósito geral.

Definição do problema

Entrada: uma lista com ‘n’ itens (em ordem crescente),

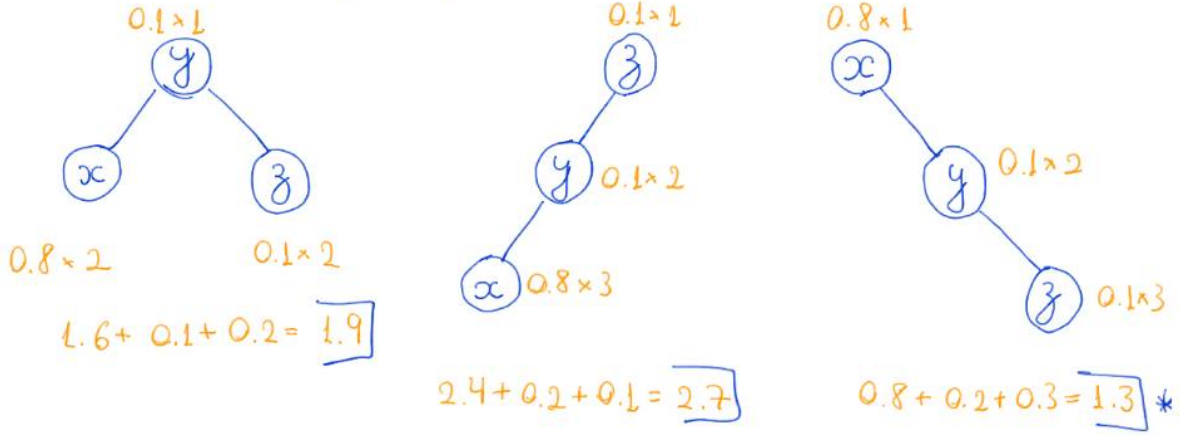
- que podem ser números, caracteres, palavras, etc,
- e uma probabilidade de acesso p_i para cada item ‘i’, com 1 <= i <= n.

Solução: uma árvore binária de busca T

- que minimiza o tempo total esperado de busca dos itens, ou seja,
  - C(T) = \sum_{i=1..n} p_i * prof(i, T)
    - sendo prof(i, T) a profundidade no item ‘i’ na árvore T.
- Lembre que profundidade é o número de nós no caminho da raiz até o item,
  - e note que o tempo de busca é proporcional à profundidade do item.

Exemplo: Dados itens x < y < z, com

- probabilidades de acesso p_x = 80%, p_y = 10% e p_z = 10%,
    - qual o tempo total esperado de busca dos itens em cada árvore?



Note que, para estas frequência de acesso a árvore balanceada não é a melhor,

- mas sim a árvore que deixa mais perto do topo o item mais provável.

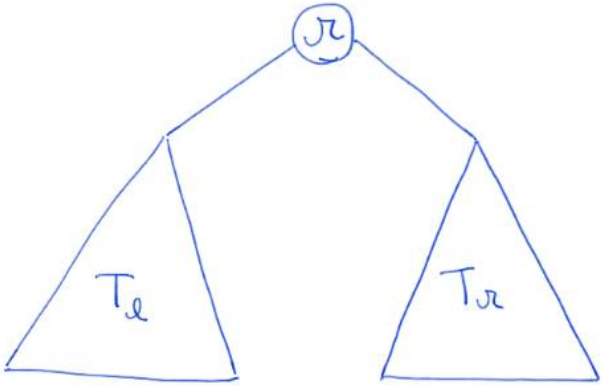
Subestrutura ótima:

Para simplificar a análise, supomos que os ‘n’ itens da nossa árvore

- tem valores distintos no conjunto {1, 2, ..., n}. Mas ressaltamos
    - que os argumentos funcionam com qualquer conjunto de itens.

Começamos pelo exercício de abstração de imaginar uma solução ótima.

- Neste caso, uma árvore binária de busca ótima T com raiz ‘r’.



O que sabemos sobre T e suas subárvores T_l e T_r?

- Como tratasse de uma árvore de busca,
    - T_l contém os itens {1, ..., r-1} e T_r contém os itens {r+1, ..., n}.
- Vamos mostrar que T_l é ótima para {1, ..., r-1} e T_r é ótima para {r+1, ..., n}.

A ideia dessa demonstração é que, se T_l ou T_r não fossem ótimas
    - para seus respectivos subconjuntos de itens,
- poderíamos obter uma árvore T* melhor que T
    - trocando suas subárvores pelas ótimas.
- Vamos formalizar este argumento, provando que T_l é ótima.

Suponha, por contradição, que T_l não é ótima, ou seja,
- que existe outra árvore T_l* para os itens {1, 2, ..., r-1}
    - com tempo total esperado de busca menor que T_l,
        - i.e., C(T_l*) < C(T_l).

Usando a definição de C(T), temos que
- C(T) = \sum_{i=1..n} p_i * prof(i, T)

- C(T_l) = \sum_{i=1.. r-1} p_i * prof(i, T_l)

- C(T_l*) = \sum_{i=1..r-1} p_i * prof(i, T_l*)

Vamos construir uma nova árvore T* para os itens {1, 2, ..., n}
    - removendo Tl de T e colocando Tl* no lugar.
- Assim, para concluir a prova, basta mostrar que C(T*) < C(T)
    - pois isso contradiz o fato de T ser ótima.

Vamos analisar C(T) mais de perto.

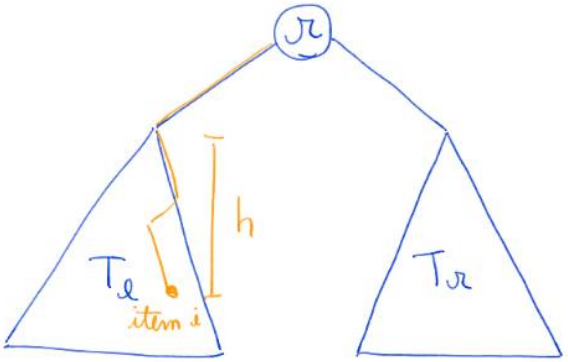
- C(T) = \sum_{i=1..n} p_i * prof(i, T)

Como T está enraizada em ‘r’, vamos dividir o somatório

  - em três blocos {1, ..., r-1}, {r} e {r+1, ..., n}.

- C(T) = \sum_{i=1..r-1} p_i * prof(i, T) + p_r * 1 + \sum_{i=r+1..n} p_i * prof(i, T)

Foque num item ‘i’ em T_l. Note que prof(i, T) = 1 + prof(i, T_l).



  - Observe que o mesmo vale para um item em Tr. Assim,

- C(T) = \sum_{i=1..r-1} p_i * (1 + prof(i, T_l)) + p_r
  + \sum_{i=r+1..n} p_i * (1 + prof(i, T_r))

Tirando os termos constantes dos somatórios

- C(T) = \sum_{i=1..r-1} p_i * prof(i, T_l) + \sum_{i=r+1..n} p_i * prof(i, T_r)
  + p_r + \sum_{i=1..r-1} p_i * 1 + \sum_{i=r+1..n} p_i * 1

Agrupando os somatórios, obtemos

- C(T) = \sum_{i=1..r-1} p_i * prof(i, T_l) + \sum_{i=r+1..n} p_i * prof(i, T_r)
  + \sum_{i=1..n} p_i

Note que os dois primeiros somatórios são C(T_l) e C(T_r). Portanto,

- C(T) = C(T_l) + C(T_r) + \sum_{i=1..n} p_i

Fazendo o mesmo desenvolvimento para C(T*) chegamos a

- C(T*) = C(T_l*) + C(T_r) + \sum_{i=1..n} p_i

Para concluir, como supusemos por absurdo que C(T_l) > C(T*_l), temos

- C(T) = C(T_l) + C(T_r) + \sum_{i=1..n} p_i
  > C(T_l*) + C(T_r) + \sum_{i=1..n} p_i = C(T*)

    - o que contraria o fato de que T é ótima.

A prova da otimalidade de T_r é idêntica.

Recorrência:

Dada a subestrutura ótima que encontramos

- uma árvore binária de busca ótima T enraizada em ‘r’ tem
    - subárvore esquerda T_l e direita T_r que são ótimas, respectivamente,
        - para os conjuntos de itens {1, ..., r-1} e {r+1, ..., n}.

Note que não sabemos qual é a raiz da árvore ótima, o que leva à questão:

- Quais as raízes possíveis para uma árvore que contém o conjunto {1, ..., n}?
    - R: São ‘n’ raízes possíveis, i.e, todos os itens em {1, ..., n}.
- Isso indica que na nossa recorrência
    - temos que escolher a melhor solução entre as ‘n’ possíveis.

Outro ponto relevante é, quando estamos considerando uma raiz ‘r’,

    - quantos/quais subproblemas temos que considerar?
- R: São dois subproblemas,
    - um com a subárvore esquerda que contém os itens {1, ..., r-1}
        - e outro com a subárvore direita que contém os itens {r+1, ..., n}.

E como se relacionam os custos da árvore T enraizada em ‘r’

    - com o custo de suas subárvores ótimas?
- R: C(T) = C(T_l) + C(T_r) + \sum_{k=1..n} p_k

Escrevendo a relação de custo de uma árvore ótima para os itens {1, ..., n}

- considerando as ‘n’ raízes possíveis e os subproblemas
    - que surgem em cada caso, temos a seguinte recorrência
- A[1, n] = min_{r=1,...,n} {\sum_{k=1..n} p_k + A[1, r-1] + A[r+1, n]}
    - com a ressalva de que A[i, j] = 0 se i > j.

Note, no entanto, que nossa recorrência está escrita com os parâmetros 1 e ‘n’.

- Será que só precisamos nos preocupar
    - com sequências de itens começadas em 1 e terminadas em ‘n’?
- R: Não. Para perceber isso, considere resolver o problema A[1, n] e note que
    - para isso precisamos do valor de A[1, j] para todo ‘j’ em [1, n].
- Foque em um ‘j’ particular e considere o subproblema A[1, j].
    - Agora, repetindo o raciocínio, para resolver A[1, j]
        - precisamos do valor de A[i, j] para todo i em [1, j].
- Portanto, nosso total de subproblemas corresponde
    - a todos os intervalos contínuos {i, i+1, ..., j-1, j} com i <= j.

Assim, reescrevendo nossa recorrência, para todo 1 <= i <= j <= n temos que

- A[i, j] = min_{r=i,...,j} {\sum_{k=i..j} p_k + A[i, r-1] + A[r+1, j]}
    - lembrando da nossa convenção de que A[i, j] = 0 se i > j.

Algoritmo: A base do algoritmo é sempre ter certeza de que

- os problemas menores serão resolvidos antes dos maiores.

Pseudocódigo:

```
algABBO(n, p):
    para tam = 1 até n:
        para i = 1 até n - tam + 1:
            j = i + tam - 1
            A[i, j] = min_{r=i,...,j} {\sum_{k=i..j} p_k + A[i, r-1] + A[r+1, j]}
    devolva A[1, n]
```

Para entender o funcionamento desse algoritmo, observe que

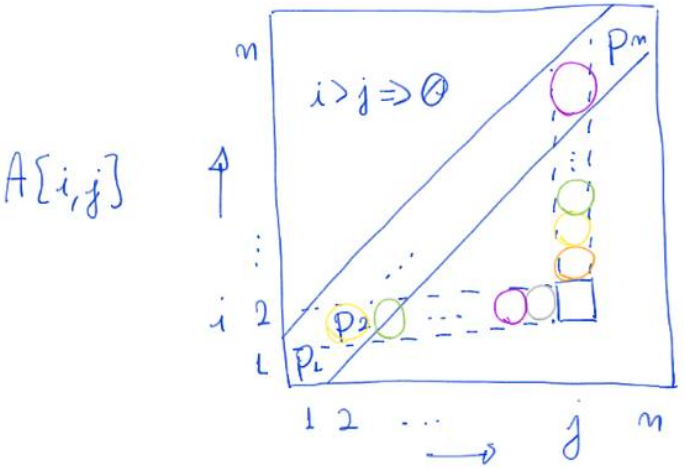
- ‘tam’ é o tamanho do subproblema/intervalo corrente de itens,
  - ‘i’ é o início de um intervalo e ‘j’ é o final do mesmo.
- Assim, começamos no menor tamanho de subproblema
  - e aumentamos a cada iteração.

Note também que o algoritmo começa preenchendo

- a diagonal principal de uma matriz,
- e em cada iteração subsequente preenche uma “diagonal” menor.

Por fim, observe que para preencher uma célula A[i, j], o algoritmo considera

- os subproblemas da linha ‘i’ e da coluna ‘j’, utilizando-os aos pares.



Eficiência: \Theta(n^3), pois tem de resolver \Theta(n^2) recorrências

- (preencher metade de uma matriz n por n),
- e para resolver cada recorrência (preencher cada célula da matriz)
  - precisa consultar em média \Theta(n) outras posições da matriz.

Construindo a árvore: demanda saber qual a raiz escolhida em cada subproblema.

- Guardar tal raiz numa matriz auxiliar B cada vez que o algoritmo
  - resolve uma recorrência torna a construção da árvore mais eficiente,
    - embora não seja obrigatório.