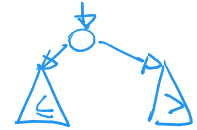
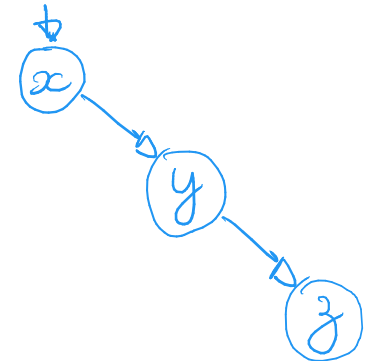
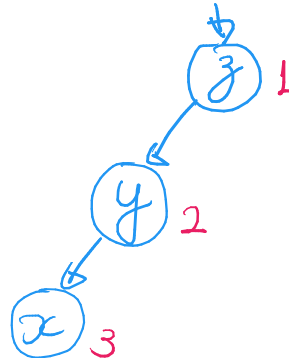
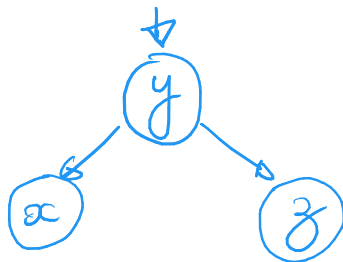


## Problema da Árvore Binária de Busca Ótima



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 no 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  $O(\lg n)$  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  $1 \leq i \leq n$

Solução: uma árvore binária de busca  $T$

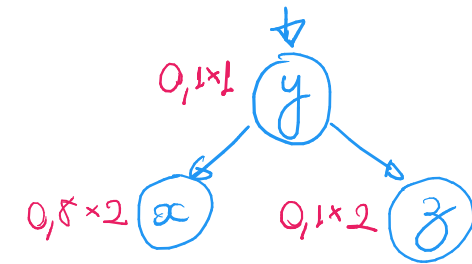
- que minimiza o tempo total esperado de busca dos itens, ou seja,

$$C(T) = \sum_{i=1}^n p_i \cdot \text{prof}(i, T)$$

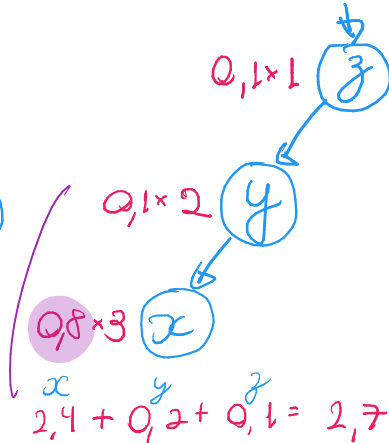
- sendo  $\text{prof}(i, T)$  a profundidade no item  $i$  na árvore  $T$
- Lembre que profundidade é o número de nós no caminho da raiz até o item,
  - e note que o tempo de busca é proporcional à profundidade do item.

Exemplo: Dados itens  $x < y < z$  com

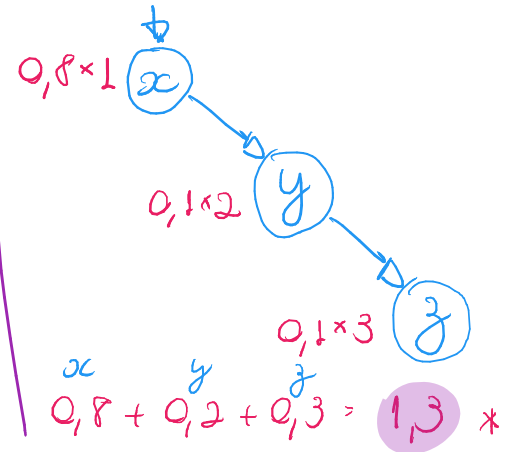
- probabilidades de acesso  $p_x = 80\%$ ,  $p_y = 10\%$  e  $p_z = 10\%$ 
  - qual o tempo total esperado de busca dos itens em cada árvore?



$$y \quad x \quad z \\ 0,1 + 1,6 + 0,2 = 1,9$$



$$z \quad y \quad x \\ 2,4 + 0,2 + 0,1 = 2,7$$



$$x \quad y \quad z \\ 0,8 + 0,2 + 0,3 = 1,3 *$$

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

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

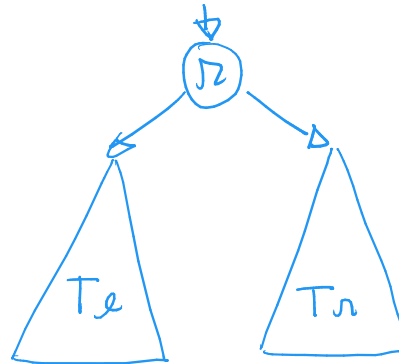
## Subestrutura ótima

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

- tem valores distintos no conjunto  $\{1, 2, \dots, n\}$  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  $r$

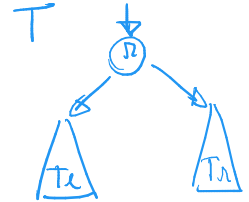


O que sabemos sobre  $T$  e suas subárvores  $T_e$  e  $T_n$ ?

- Como temos uma árvore binária de busca,
  - $T_e$  contém os itens  $\{1, \dots, a-1\}$  e  $T_n$  contém os itens  $\{a+1, \dots, n\}$
- Vamos mostrar que  $T_e$  é ótima para  $\{1, \dots, a-1\}$  e  $T_n$  é ótima para  $\{a+1, \dots, n\}$

A ideia dessa demonstração é que, se  $T_e$  ou  $T_n$  não fossem ótimas

- para seus respectivos subconjuntos de itens,
- poderíamos obter uma árvore  $T^*$  melhor que  $T$ 
  - trocando sua subárvore esquerda e/ou direita pelas ótimas.
- Vamos formalizar este argumento, provando que  $T_e$  é ótima.



Suponha, por contradição, que  $T_e$  não é ótima, ou seja,

- que existe outra árvore  $T_e^*$  para os itens  $\{1, 2, \dots, n-1\}$ 
  - com tempo total esperado de busca menor que
    - i.e.,  $[C(T_e^*) < C(T_e)] \Leftarrow$  por absurdo

Usando a definição de  $C(T)$  temos

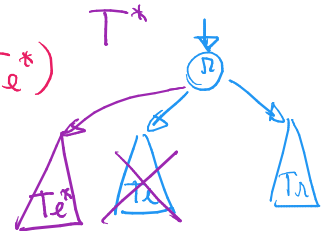
$$C(T) = \sum_{i=1}^n p_i \cdot \text{prof}(i, T)$$

$$C(T_e) = \sum_{i=1}^{n-1} p_i \cdot \text{prof}(i, T_e) \quad \textcircled{a}$$

$$C(T_e^*) = \sum_{i=1}^{n-1} p_i \cdot \text{prof}(i, T_e^*)$$

Vamos construir uma nova árvore  $T^*$  para os itens  $\{1, 2, \dots, n\}$

- removendo  $T_e$  de  $T$  e colocando  $T_e^*$  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.

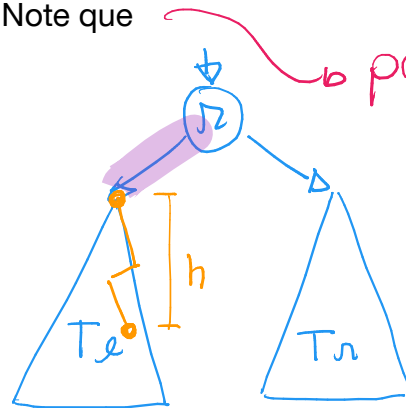
$$C(T) = \sum_{i=1}^n p_i \cdot \text{prof}(i, T)$$

Como  $T$  está enraizada em  $n$  vamos dividir o somatório

- em três blocos

$$\Rightarrow C(T) = \sum_{i=1}^{n-1} p_i \cdot \text{prof}(i, T) + p_n \cdot 1 + \sum_{i=n+1}^n p_i \cdot \text{prof}(i, T)$$

Foque num item  $i$  em  $T_e$ . Note que



$$\text{prof}(i, T) = 1 + \text{prof}(i, T_e) \quad *1$$

$p / i \in T_e$

$$\text{prof}(i, T) = 1 + \text{prof}(i, T_n) \quad *2$$

$p / i \in T_n$

- Observe que o mesmo vale para um item em  $T_n$ . Assim

$$C(T) = \sum_{i=1}^{n-1} p_i (1 + \text{prof}(i, T_e)) + p_n + \sum_{i=n+1}^n p_i (1 + \text{prof}(i, T_n))$$

Tirando os termos constantes dos somatórios

$$C(T) = \sum_{i=1}^{n-1} p_i \cdot 1 + \sum_{i=1}^{n-1} p_i \cdot \text{prof}(i, T_e) + p_n + \sum_{i=n+1}^n p_i \cdot 1 + \sum_{i=n+1}^n p_i \cdot \text{prof}(i, T_n)$$

Agrupando os somatórios, obtemos

$$C(T) = \sum_{i=1}^{n-1} p_i \cdot \text{prof}(i, T_e) + \sum_{i=n+1}^n p_i \cdot \text{prof}(i, T_n) + \sum_{i=1}^n p_i$$

Note que os dois primeiros somatórios são  $C(T_e)$  e  $C(T_n)$ . Portanto,

$$\Rightarrow C(T) = c(T_e) + c(T_n) + \sum_{i=1}^n p_i$$

Fazendo o mesmo desenvolvimento para  $T^*$  chegamos a

$$c(T^*) = c(T_e^*) + c(T_n) + \sum_{i=1}^n p_i$$

Para concluir, como supusemos por absurdo que  $[c(T_e^*) < c(T_e)]$ , temos

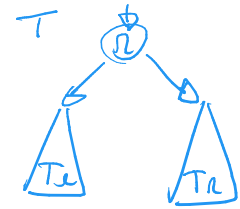
$$[C(T) = c(T_e) + c(T_n) + \sum_{i=1}^n p_i > c(T_e^*) + c(T_n) + \sum_{i=1}^n p_i = c(T^*)]$$

- Quer dizer, se  $T_e$  não for ótima, então  $T$  também não é,
  - chegamos em uma contradição.

A prova da otimalidade de  $T_n$  é idêntica.



## Recorrência



Dada a subestrutura ótima que encontramos

- uma árvore binária de busca ótima  $T$  enraizada em  $n$  tem
  - subárvore esquerda  $T_l$  e direita  $T_r$  que são ótimas,
- respectivamente, para os conjuntos de itens  $\{1, \dots, a-1\}$  e  $\{a+1, \dots, n\}$

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  $\{1, \dots, n\}$ ?
  - R: São  $n$  raízes possíveis, i.e, todos os itens em  $\{1, \dots, n\}$
- Isso indica que na nossa recorrência
  - temos que escolher a melhor solução entre as  $n$  possíveis.

Outro ponto relevante é, quando fixamos uma raiz  $n$

- quantos/quais subproblemas temos que considerar?
- R: São dois subproblemas,
  - um que contém os itens  $\{1, \dots, a-1\}$ 
    - e outro que contém os itens  $\{a+1, \dots, n\}$

E como se relacionam os custos da árvore  $T$  enraizada em  $n$

- 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  $\{1, \dots, n\}$

- considerando as  $n$  raízes possíveis e
  - os subproblemas de cada caso, temos a recorrência

$$A[1, n] = \min_{r=1, \dots, n} \left\{ \sum_{k=1}^r p_k + A[1, r-1] + A[r+1, n] \right\}$$

$O(n)$

- com a ressalva de que  $A[i, j] = 0$  se  $i > j$

Note, no entanto, que nossa recorrência está escrita com os parâmetros  $l$  e  $n$

- Será que só precisamos nos preocupar
  - com sequências de itens começadas em  $l$  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  $A[1, i]$  para todo  $i$  em  $[1, j]$
- Portanto, nosso total de subproblemas corresponde
  - a todos os intervalos contínuos  $[i, i+1, \dots, j]$  com  $i \leq j$

$$A[1, n] = A[1, r-1] + \dots$$

$$A[1, r-1] = A[1, r-2] + \dots$$

$$+ \sum p_k + A[r+1, n-1]$$

$O(n^2)$

Assim, reescrevendo nossa recorrência, para todo par  $1 \leq i < j \leq n$  temos

$$A[i, j] = \min_{r=i \dots j} \left\{ \sum_{k=i}^r p_k + A[i, r-1] + A[r+1, j] \right\}$$

- 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.

algABBO(n, p):

para todo par  $(i, j) \forall i > j: A[i, j] = 0$

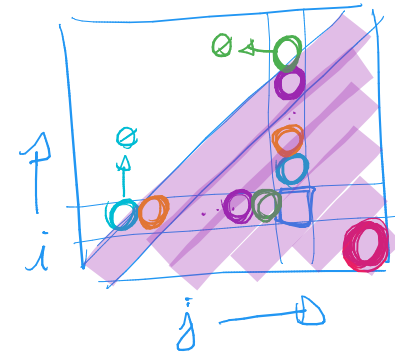
para  $\text{tam} = 1$  até  $n$ :

para  $i = 1$  até  $n - \text{tam} + 1$ :

$j = i + \text{tam} - 1$

$$A[i, j] = \min_{r=i..j} \left\{ \sum_{k=i}^r P_k + \underbrace{A[i, r-1]}_{1^\circ \text{ s.p.}} + \underbrace{A[r+1, j]}_{2^\circ \text{ s.p.}} \right\}$$

devolve  $A[1, n]$



2 laços  
aninhados  
realizam  
 $\Theta(n^2)$   
iterações

Para entender o funcionamento desse algoritmo, observe que

- $\text{tam}$  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.

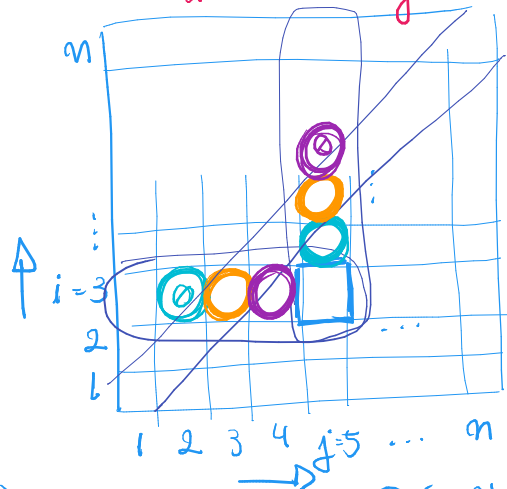
Note também que o algoritmo começa preenchendo

- 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.

$A[i,j]$



$$A[i,j] = \min_{r=i..j} \left\{ \sum_{k=i}^j p_k + \underbrace{A[i,r-1]}_{1^\circ \text{ s.p.}} + \underbrace{A[r+1,j]}_{2^\circ \text{ s.p.}} \right\}$$

$$r = 3$$

$$r = 4$$

$$r = 5$$

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