Problem Solving Paradigms

Divide & Conquer; Greedy

André, Salles

Departamento de Informática
Universidade Federal de Viçosa

INF 333 - 2022/2

Problem Solving Paradigms

- Complete Search
- Divide & Conquer
- Greedy
- Dynamic Programming

Complete Search

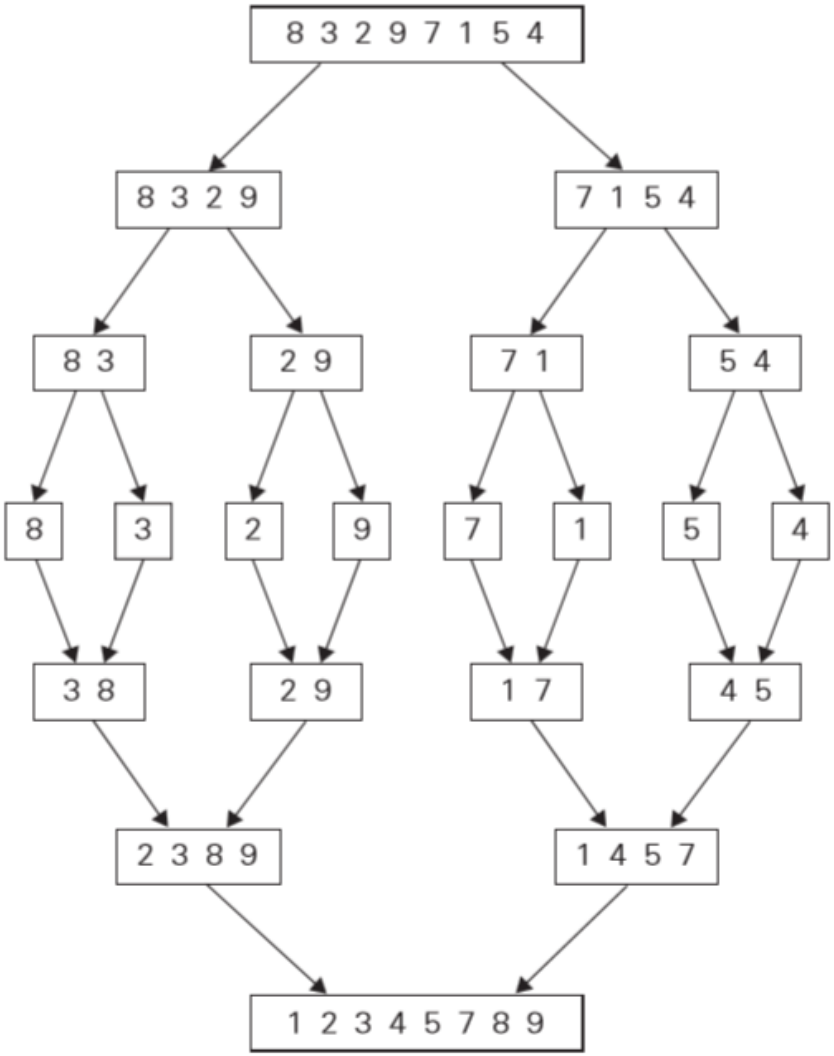
- Busca Exaustiva
- Força Bruta
- *Backtracking*

Obs.: já visto na semana anterior

Divide & Conquer

- Tenta simplificar a solução de um problema dividindo-o em partes menores e conquistando-as
- Passos
  - Dividir (+/- no meio) o problema original em *sub*-problemas
  - Encontrar *sub*-soluções para cada *sub*-problema (agora +fáceis)
  - Se necessário, combinar as *sub*-soluções para produzir uma solução completa para o problema principal
- Exemplos
  - Ordenação: quicksort, mergesort, heapsort
  - Estruturas: árvore binária de pesquisa, heap, segment tree
  - Pesquisa Binária

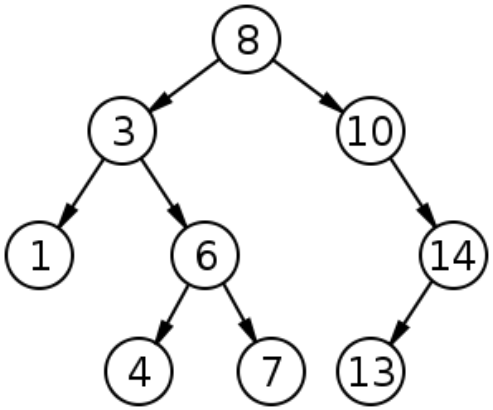
Divide & Conquer - Mergesort



Divide & Conquer - Quicksort

0	1	2	3	4	5	6	7
	i						*j*
5	3	1	9	8	2	4	7
			i			*j*	
5	3	1	9	8	2	4	7
			i			*j*	
5	3	1	4	8	2	9	7
				i	*j*		
5	3	1	4	8	2	9	7
				i	*j*		
5	3	1	4	2	8	9	7
				j	*i*		
5	3	1	4	2	8	9	7
2	3	1	4	**5**	8	9	7
	i		*j*				
2	3	1	4				
	i	*j*					
2	3	1	4				
	i	*j*					
2	1	3	4				
	j	*i*					
2	1	3	4				
1	**2**	3	4				
1							
			i j				
		3	4				
		j	*i*				
		3	4				
			4				
						i	*j*
					8	9	7
						i	*j*
					8	7	9
						j	*i*
					8	7	9
					7	**8**	9
					7		
							9

Divide & Conquer - Binary Search Tree



Divide & Conquer - Pesquisa Binária

Uso comum: buscar item em array ordenado

- Verificar se o elemento do meio é o que estamos buscando
- Se for, ou se não há mais dados, fim
- Senão, buscar na parte esquerda ou direita, conforme o caso

Complexidade

- $O(\log n)$: pois o tamanho do espaço de busca é dividido ao meio a cada verificação

Funções prontas

- C++: `binary_search` `<algorithm>`
- C: `bsearch` `<cstdlib>`

Divide & Conquer - Pesquisa Binária

Outras funções úteis

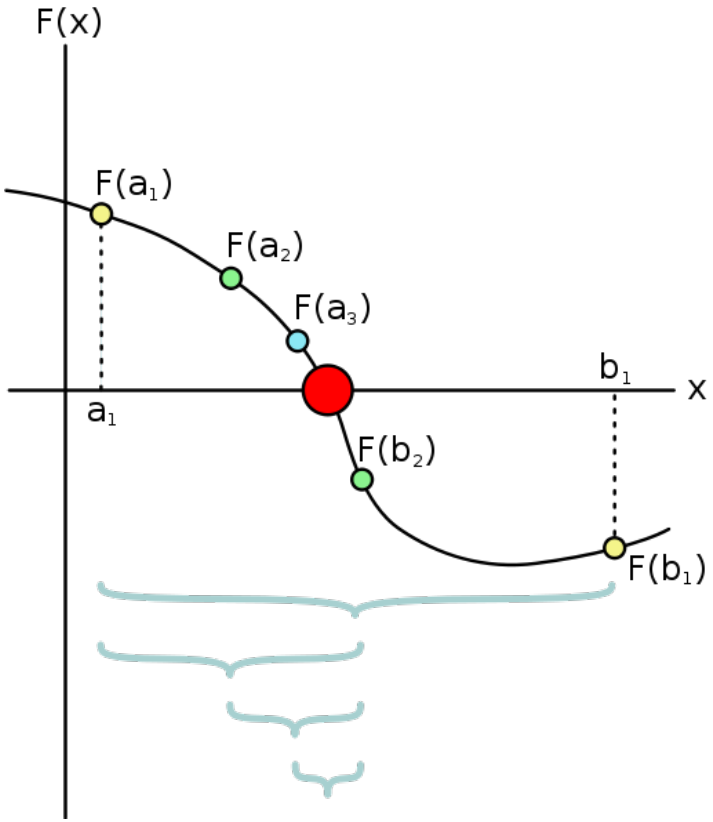
- C++: `lower_bound` `<algorithm>`
  Retorna iterador para o primeiro elemento de [*first*,*last*) que é pelo menos *x*
- C++: `upper_bound` `<algorithm>`
  Retorna iterador para o primeiro elemento de [*first*,*last*) que é maior que *x*

- C++: `equal_range` `<algorithm>`
  Retorna uma *pair* com os dois acima (intervalo com iguais a *x*)

Divide & Conquer - Bisseção

Outro uso do mesmo princípio: método da bisseção

- Buscar raiz de uma função
- Interessante quando é difícil calculá-la analiticamente
- Condição: raiz em $[a, b]$ onde $f(a)$ e $f(b)$ tem sinais opostos



Divide & Conquer - Busca Binária

Busca binária (bisseção)

Você quer fazer um empréstimo para comprar um carro e quer pagar prestações fixas de d dólares durante m meses. O valor original do carro é v dólares e o banco cobra i% do valor devido ao final de cada mês. Qual o valor de d? (ou seja, quanto deverá pagar por mês?*)

* *arredondado para 2 casas decimais*

Exemplo

Seja $d = 576$, $m = 2$, $v = 1000$, e $i = 10\%$

- Dívida após o 1º mês: $1000 \times 1.1 - 576 = 524$
- Dívida após o 2º mês: $524 \times 1.1 - 576 \approx 0$

Divide & Conquer - Busca Binária

Exemplo

Mas e se fossem dados apenas $m = 2$, $v = 1000$, e $i = 10\%$.

- Como determinar que $d = 576$?
- Em outras palavras, buscar a raiz d tal que $f(d, 2, 1000, 10) \approx 0$

Resolvendo por busca binária (bisseção)

- Primeiramente, devemos encontrar um intervalo razoável $[a \ldots b]$
- $a = 1$ já que temos que pagar alguma coisa (U$1 pelo menos)
- $b = 1100$, no caso de pagar o empréstimo depois de um 1 mês
- E então encontrar d neste intervalo por bisseção

Divide & Conquer - Busca Binária

Resolvendo por busca binária (bisseção)

- Pagando $d = (a + b)/2 = (1 + 1100)/2 = 550.5$ dólares por mês *faltarão* 53.95 depois de 2 meses.

  Então devemos *aumentar d*
- Pagando $d = (a + b)/2 = (550.5 + 1100)/2 = 825.25$ por mês *sobrarão* 523.025 depois de 2 meses. Então devemos *diminuir d*
- Pagando $d = (a + b)/2 = (550.5 + 825.25)/2 = 687.875$ por mês *sobrarão 234.5375* depois de 2 meses. Então devemos *diminuir d*
- ... poucas* iterações logarítmicas depois...
- Pagando $d = 576.190476...$ por mês, terminamos de pagar o empréstimo em 2 meses

** mais precisamente $O(log_2((b - a)/\varepsilon))$ iterações, sendo ε a tolerância de erro*

Divide & Conquer - Busca Binária

- No exemplo, $log_2 1099/\varepsilon$ iterações
- Para $\varepsilon = 1e\text{-}9$ são ≈ 40
- E para $\varepsilon = 1e\text{-}15$ são $\approx 60^*$
- muito mais eficiente que uma busca linear em todos os valores de $d = [1 \ldots 1100]/\varepsilon$

* *na verdade, alguns competidores usariam laço de 100 iterações, que garante terminação, em vez de testar erro menor que ε, que pode levar a loop infinito por erro de ponto flutuante*

Divide & Conquer - Busca binária da resposta

Exemplo: UVa 714 - Copying Books (link)

- São dados M livros numerados $1, 2, \ldots, M$
- Os livros possuem $p_1, p_2, \ldots, p_m$ páginas respectivamente
- Todos devem ser copiados; você os distribui entre K escribas ($K \leq M$)
- Cada livro é dado a apenas um escriba
- Cada escriba recebe uma sequência *consecutiva* de livros
- Ou seja, existe uma sucessão crescente de inteiros
  $0 = b_0 < b_1 < b_2 \cdots < b_{k-1} \leq b_k = m$
  tal que o i-ésimo escriba recebe os livros de $b_{i-1} + 1$ até b_i
- O tempo gasto para copiar todos os livros é determinado pelo escriba que recebeu mais trabalho
- Seu objetivo é minimizar o máximo número de páginas de um escriba

Divide & Conquer - Busca binária da resposta

- Existe uma solução por Programação Dinâmica
- Mas é possível “chutar” a resposta no estilo busca binária!

Divide & Conquer - Busca binária da resposta

Exemplo: UVa 714 - Copying Books (link)

- Suponha $M = 9$ e $K = 3$ e $p_1, p_2, \ldots, p_9$ = 100, 200, 300, 400, 500, 600, 700, 800, 900 respectivamente
- Se chutarmos uma resposta = 1000, o problema é bem mais fácil: se o escriba com mais trabalho pode copiar até 1000 páginas, é possível?
- A resposta é NÃO.
  - escriba 1: $\{100, 200, 300, 400\}^*$
  - escriba 2: $\{500\}$
  - escriba 3: $\{600\}$
  - $\{700, 800, 900\}$ para ninguém
- Então tem que ser mais que 1000

* *atribuindo de forma "gulosa", o máximo que for possível*

Divide & Conquer - Busca binária da resposta

Exemplo: UVa 714 - Copying Books

- Se chutarmos uma resposta = 2000: se o escriba com mais trabalho pode copiar até 2000 páginas, é possível?
- A resposta é SIM, e com sobra!
  - escriba 1: $\{100, 200, 300, 400, 500\}$
  - escriba 2: $\{600, 700\}$
  - escriba 3: $\{800, 900\}$
- Os escribas 1, 2, 3 têm sobra respectivamente de $\{500, 700, 300\}$
- Então pode ser menos que 2000

Fazer busca binária no intervalo $[1, \sum_{i=1}^{m} p_i]$

Divide & Conquer - generalização

Encontrar a posição onde o valor de uma função muda.

- Seja `ok(x)` uma função que returna `true` se x é uma solução válida, `false` senão;
- E que `ok(x)` é `false` quando $x < k$ e `true` quando $x \geq k$.

x	0	1	...	$k-1$	k	$k+1$	...
`ok(x)`	`false`	`false`	...	`false`	`true`	`true`	...

Iniciar com um valor z alto o suficiente que se sabe ser `true`.
A função `ok(x)` será chamada $O(\log z)$ vezes.

Implementando buscas binárias

- Busca binária pronta do C++: pesquisar em array (precisamos de algo mais genérico).
- Para usar busca binária em funções/predicados: precisa ser monotonica (*false, false, false, true, true true* OU *true true true, false, false, false*)
  - Tente se convencer/provar que seu problema é assim.
  - Tente pensar: em problemas de maximização/minimização ou similares, se eu tiver um valor K, consigo descobrir se esse valor "é ok" ? (exemplo: 1000 páginas do livro por escriba é uma solução válida?)

Implementando buscas binárias

- Tente modelar o problema como um de decisão
- TSP: encontrar o menor ciclo
- TSP de decisão: predicado *Existe um ciclo com custo* $\leq X$ *?* (no caso do TSP esse predicado é difícil de criar, mas em muitos problemas é fácil)
- Busca de elemento em array: onde está o número X?
- Busca de elemento em array (decisão): considerando a posição i, temos $A[i] \leq X$?
  - A=[1,2,5,9,10]
  - Temos $A[i] \leq 8$? [true,true,true,false,false]
  - Temos $A[i] \leq 9$? [true,true,true,false,false]
  - Isso é justamente o que a função lower_bound do C++ verifica.

Implementando buscas binárias

- Veja exemplos de código para busca binária em binaria.cpp
- Idealmente, não mude esse código. Tente adaptar apenas o predicado.
- Algoritmos implementados com base em: Tutorial de busca binária do TopCoder. Nesse link há um EXCELENTE tutorial de busca binária.
- Exercício: tente resolver alguns problemas com esse código.

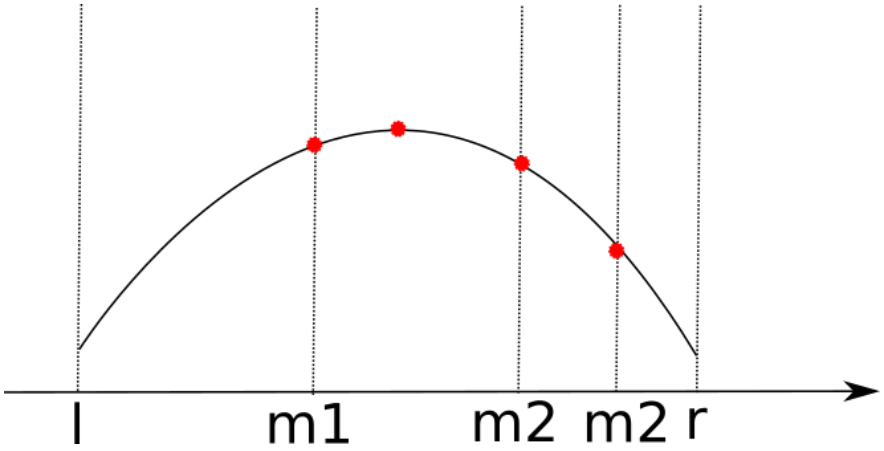
Implementando buscas binárias

- C++ tem funções que permitem busca binária em intervalos de iteradores (binary_search, lower_bound, upper_bound, partition_point)
- Elas suportam predicados
- Desafio: precisamos de iteradores (normalmente são utilizados para estruturas de dados)
- C++20: agora temos iteradores de *ranges* (intervalos de inteiros – igual range() de Python)
- Notícia ruim: nem g++9 suporta isso ainda... arriscado usar na maratona (calouros terão vida mais fácil...).
- Vejam: https://codeforces.com/blog/entry/97061

Busca ternária

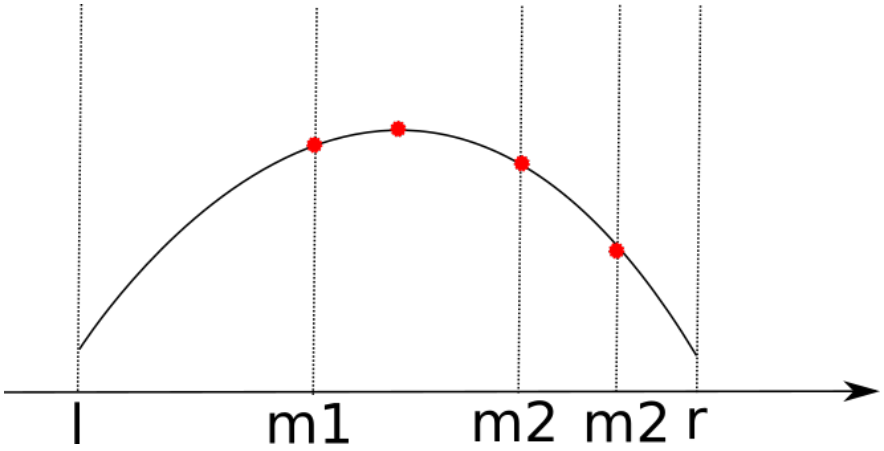
- Para funções "unimodais" (sobe de forma estrita, atinge um máximo e depois desce – ou vice-versa)
- Exemplo de utilidade: achar o máximo de uma função contínua (mas pode ser utilizado com valores discretos também).
- Para ilustrar, vamos tentar encontrar o máximo de uma função unimodal no intervalo [l,r]
- Exemplo de implementação aqui (link)
- Ideia resumida: intervalo dividido em 3 sub-intervalos de mesmo tamanho. Máximo nunca estará *no subintervalo extremo ao lado do menor*

Busca ternária



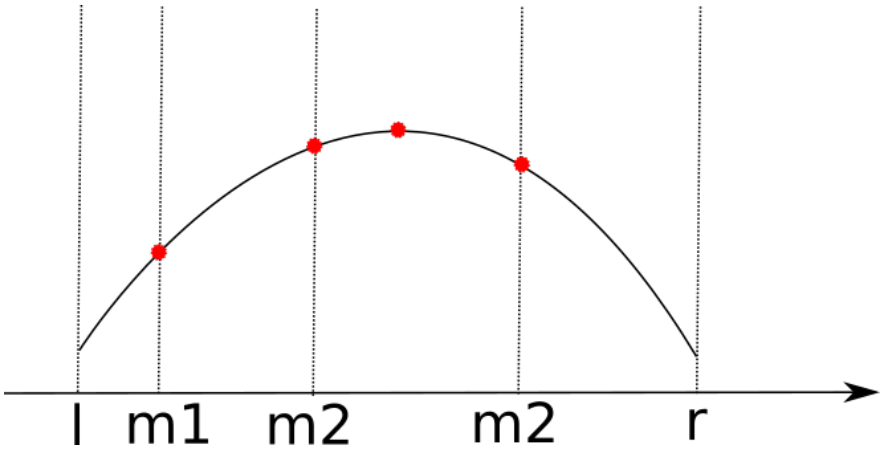
- Para ilustrar, vamos tentar encontrar o máximo de uma função unimodal no intervalo [l,r]
  - Pegamos dividimos [l,r] em tres intervalos, com os pontos m_1 e m_2 definindo os subintervalos (de mesmo tamanho – na figura não são de mesmo tamanho).
  - Calculamos f(m_1) e f(m_2)
  - Se f(m_1) $>$ f(m_2)

Busca ternária



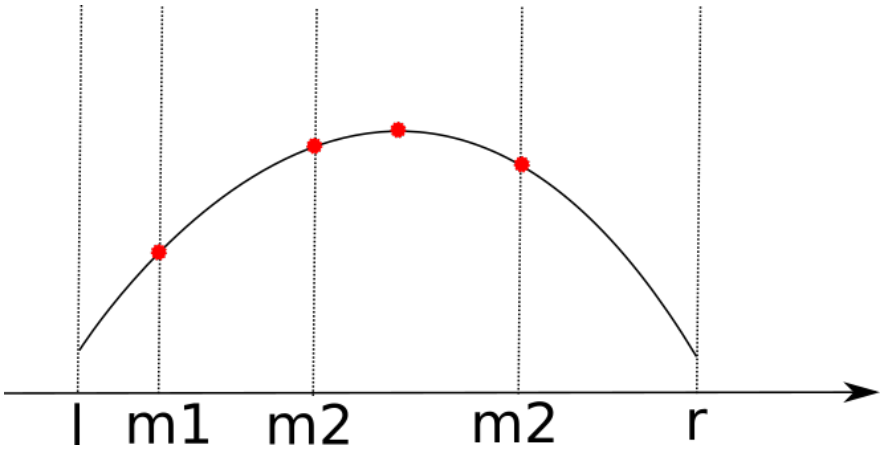
- Para ilustrar, vamos tentar encontrar o máximo de uma função unimodal no intervalo [l,r]
  - Pegamos dividimos [l,r] em tres intervalos, com os pontos m_1 e m_2 definindo os subintervalos (de mesmo tamanho – na figura não são de mesmo tamanho).
  - Calculamos f(m_1) e f(m_2)
  - Se f(m_1) $>$ f(m_2)
  - $\rightarrow$ Com certeza podemos descartar [m_2,r]

Busca ternária



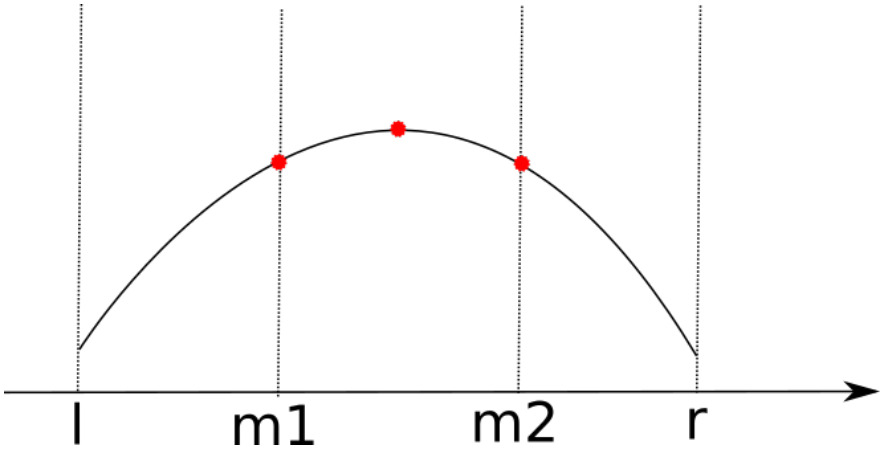
- Para ilustrar, vamos tentar encontrar o máximo de uma função unimodal no intervalo [l,r]
  - Pegamos dividimos [l,r] em tres intervalos, com os pontos m_1 e m_2 definindo os subintervalos (de mesmo tamanho – na figura não são de mesmo tamanho).
  - Calculamos f(m_1) e f(m_2)
  - Se f(m_1) $<$ f(m_2)

Busca ternária



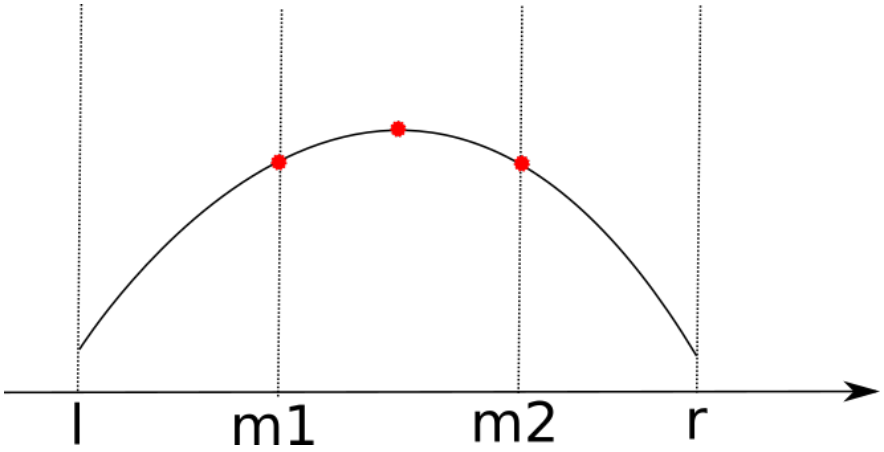
- Para ilustrar, vamos tentar encontrar o máximo de uma função unimodal no intervalo [l,r]
  - Pegamos dividimos [l,r] em tres intervalos, com os pontos m_1 e m_2 definindo os subintervalos (de mesmo tamanho – na figura não são de mesmo tamanho).
  - Calculamos f(m_1) e f(m_2)
  - Se f(m_1) $<$ f(m_2)
  - $\rightarrow$ Com certeza podemos descartar [l,m_1]

Busca ternária



- Para ilustrar, vamos tentar encontrar o máximo de uma função unimodal no intervalo [l,r]
  - Pegamos dividimos [l,r] em tres intervalos, com os pontos m_1 e m_2 definindo os subintervalos (de mesmo tamanho – na figura não são de mesmo tamanho).
  - Calculamos f(m_1) e f(m_2)
  - Se f(m_1) $=$ f(m_2)

Busca ternária



- Para ilustrar, vamos tentar encontrar o máximo de uma função unimodal no intervalo [l,r]
  - Pegamos dividimos [l,r] em tres intervalos, com os pontos m_1 e m_2 definindo os subintervalos (de mesmo tamanho – na figura não são de mesmo tamanho).
  - Calculamos f(m_1) e f(m_2)
  - Se f(m_1) = f(m_2)
  - $\rightarrow$ O máximo com certeza está em $[m_1, m_2]$
  - Podemos tratar isso separadamente (eliminando os dois intervalos extremos) ou junto com os dois casos acima (eliminando apenas um extremo)

Busca ternária

- Dica: tenha código pronto para busca ternária em números reais e inteiros.

Busca ternária

Busca ternária

- Para inteiros, podemos utilizar simplesmente uma binária para encontrar máximo/mínimo: basta usar o predicado $p(x) = f(x) < f(x+1)$
- Dica: busca ternária em inteiros usando binária (link)

Decomposição

- Existem “poucos” algoritmos básicos usados em campeonatos de programação
- Problemas mais difíceis envolvem a combinação de 2 (ou mais) deles
- Tente decompor o problema em partes e resolvê-las de forma independente, cada uma com o método apropriado

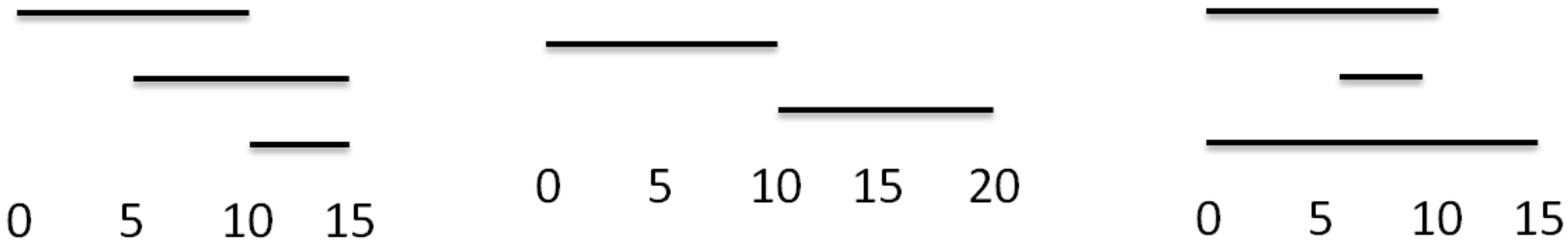
Decomposição

Exemplo: ICPC 4445 - A Careful Approach (link)

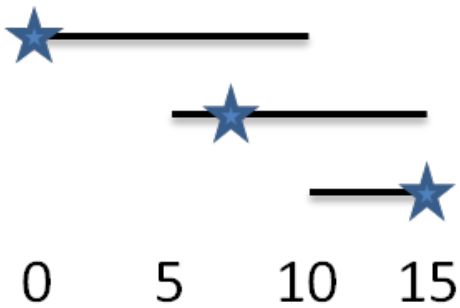
São dados cenários de aterrissagem. Cada um tem N aviões ($2 \leq N \leq 8$). Cada avião tem uma janela de tempo durante a qual é seguro aterrissar. Assim, o avião i deve aterrissar no intervalo $[a_i, b_i]$. ($0 \leq a_i \leq b_i \leq 1440$)
Tarefa:

1. Encontrar uma **ordem de aterrissagem** para todos os aviões que respeite as janelas de tempo
2. Aterrissagens devem ser distantes **tanto quanto possível** de tal forma que o menor intervalo entre duas consecutivas seja o maior possível
3. Imprimir a resposta em minutos e segundos, arredondada para o segundo mais próximo

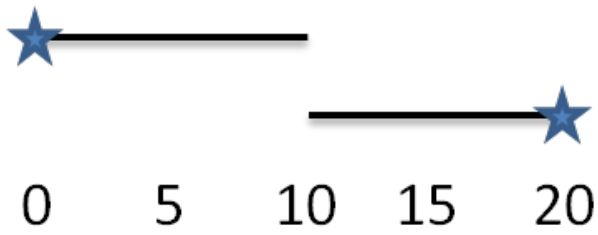
Decomposição



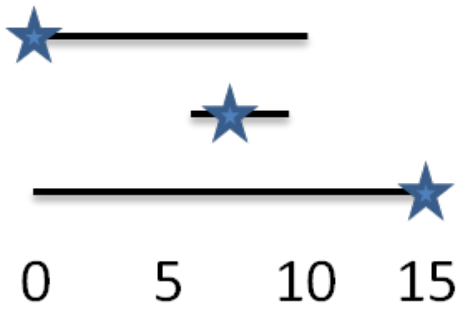
Decomposição



Resposta: 7:30



Resposta: 20:00



Resposta: 7:30

Decomposição

Solução

1. São 8 aviões. A melhor ordem pode ser achada por **Força Bruta**
2. Para cada ordem, seja L o maior intervalo de tempo.
   Verificar se L é viável pode ser feito por **Método Greedy**:
   - O primeiro avião aterrissa o mais cedo possível: a_i
   - Demais aviões: $max(a_i, \text{horário do anterior} + L)$
3. Para intervalo L muito pequeno/grande
   - o último avião aterrissa antes/depois de b_i
   - então o valor de L deve ser aumentado/diminuído
   - encontrar o melhor valor por **Divide & Conquer**

Dynamic Programming

Obs.: será visto na próxima semana