Algoritmos e Estruturas de Dados

Marcelo Lobosco
DCC/UFJF

Programação Dinâmica

Aula 07

Agenda

- Programação Dinâmica
  - Introdução
  - Exemplos
    - Fibonacci
    - Problema do Troco
    - Problema da Mochila

Introdução

- Quatro paradigmas mais comumente utilizados no desenvolvimento de aplicações
  - Backtraking (busca completa)
  - Divisão e conquista
  - Busca gulosa
  - Programação dinâmica

Backtracking

- Busca sistemática por todas as soluções possíveis de um problema
- Ao enumerar todas as possibilidades, garante que solução será encontrada: força bruta
- Uma dada solução é testada uma única vez
- Pode-se empregar métodos para melhorar desempenho, descartando soluções que não levem a solução desejada (*prunning*)

Backtracking

- Algoritmo Genérico para backtracking
  - Solução modelada como arranjo (A_1, A_2, …, A_n)
  - Cada elemento A_i do arranjo representa uma parte na solução
  - A cada passo, tem-se uma solução parcial com K elementos (A_1, A_2, …, A_k)
  - Testa-se se a solução desejada foi encontrada
  - Se não foi, tenta-se chegar a solução pela inclusão de mais um elemento ao vetor (A_1, A_2, …, $A_{k,}$ A_{k+1})

Backtracking

- Implementação

```
finished ← FALSE
BACKTRACK(A,k,input)
    if IS_A_SOLUTION(A,k,input)
        PROCESS_SOLUTION(A,k,input)
    else
        k ← k+1
        CONSTRUCT_CANDIDATES(A,k,input,c,ncandidates)
        for i ←1 to ncandidates do
            A[k] ← C[i]
            BACKTRACK(A,k,input)
            if (finished) return
```

Backtracking

- Rotina IS_A_SOLUTION verifica se primeiros k elementos de A levam a solução desejada
  - Parâmetro input permite passar dados específicos de um problema para rotina
- Rotina CONSTRUCT_CANDIDATES preenche arranjo c com conjunto completo de candidatos para ocupar posição k do arranjo A, dado conteúdo das k-1 posições
- Rotina PROCESS_SOLUTION imprime, conta ou realiza processamento necessário, dado que solução foi encontrada

Backtracking

- Implementação 1: subconjuntos

```
void process_solution(int a[], int k, data input) {
  int i; /* counter */
  printf("{");
  for (i=1; i<=k; i++)
    if (a[i] == true) printf(" %d",i);
  printf(" }\n");
}
int is_a_solution(int a[], int k, int n) { return (k == n) };
```

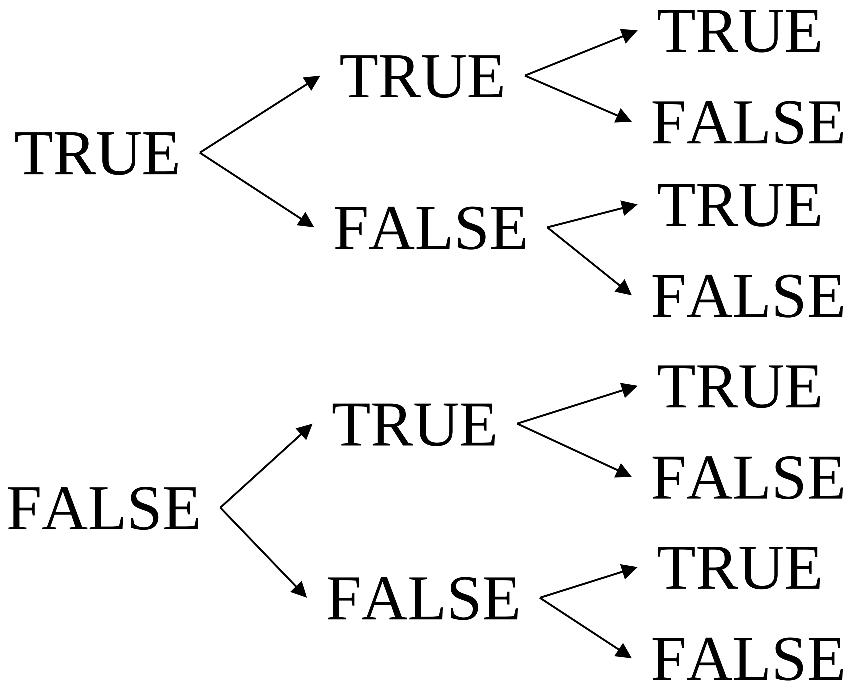
Backtracking

- Implementação 1: subconjuntos

```
void construct_candidates(int a[], int k, int n, int c[], int
   *ncandidates) {
  c[0] = true;
  c[1] = false;
  *ncandidates = 2;
}
int main (void) {
        int a[NMAX];                    /* solution vector */
        backtrack(a,0,3);
}
```

Backtracking

- Vetor c para cada chamada à construct_candidates



Backtracking

- Implementação 2: permutação

```
void process_solution(int a[], int k, data input) {
        int i;                          /* counter */
        for (i=1; i<=k; i++) printf(" %d",a[i]);
        printf("\n");
}

int is_a_solution(int a[], int k, int n) { return (k == n) };
```

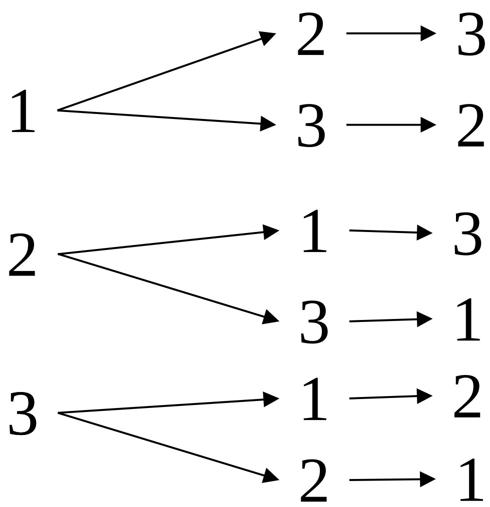
Backtracking

- Implementação 2: permutação

```
void construct_candidates(int a[], int k, int n, int c[], int *ncandidates) {
	int i;				/* counter */
	bool in_perm[NMAX];		/* what is now in the permutation? */
	for (i=1; i<NMAX; i++) in_perm[i] = false;
	for (i=1; i<k; i++) in_perm[ a[i] ] = true;
	*ncandidates = 0;
	for (i=1; i<=n; i++)
		if (in_perm[i] == false) {
			c[ *ncandidates] = i;
			*ncandidates = *ncandidates + 1;
		}
}
```

Backtracking

- Vetor c para cada chamada à construct_candidates



Programação Dinâmica

- Resolve problemas pela combinação das soluções de subproblemas
  - Semelhante a abordagem dividir para conquistar
- Ponto chave: PD utilizada quando há sobreposição de subproblemas
  - Dividir para conquistar repetidamente resolve subproblemas repetidos
  - PD resolve apenas uma vez cada subproblema

Programação Dinâmica

- Para evitar resolver repetidamente um mesmo subproblema, utiliza-se uma tabela
  - Tabela armazena resultados já computados de subproblemas resolvidos, evitando resolvê-los novamente

Programação Dinâmica

- Tipicamente aplica-se PD em problemas de otimização
  - Pode ter muitas soluções possíveis
  - Cada solução tem um valor
  - Desejamos achar uma solução ótima (maior ou menor valor)
  - Usamos termo “uma solução ótima” ao invés de “a solução ótima” visto que podem existir inúmeras soluções que possuam o valor ótimo

Programação Dinâmica

- Exemplo 1: Fibonacci
- Sequência de Fibonacci: 1, 1, 2, 3, 5, 8, …
- De modo geral, podemos dizer que sequencia é calculada seguindo a seguinte fórmula:
  - $F_0 = 1$
  - $F_1 = 1$
  - $F_n = F_{n-1} + F_{n-2}$

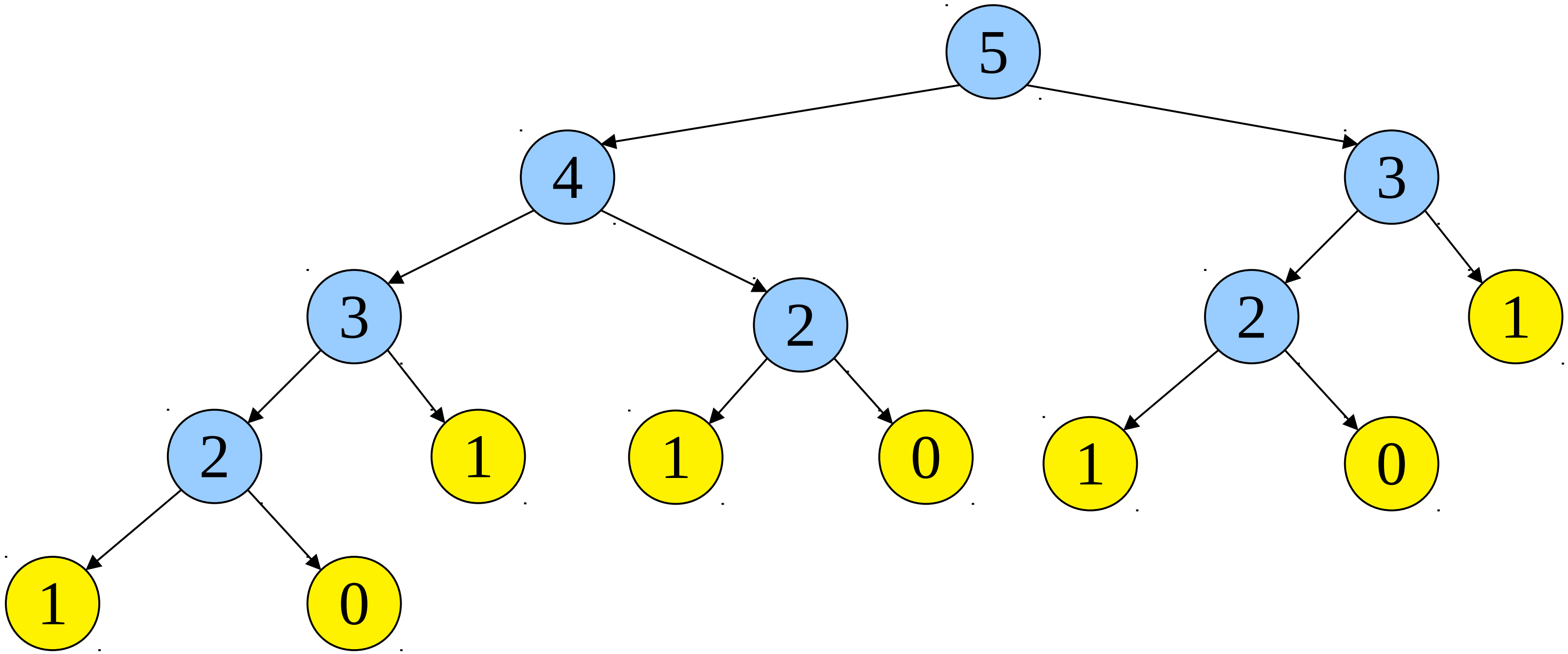
Programação Dinâmica

- Implementação tradicional:

```
int fibonacci (int n) {
    if (n <= 1) return 1;
    return fibonacci (n-1) + fibonacci (n-2);
}
```

Programação Dinâmica

- Árvore de recursão

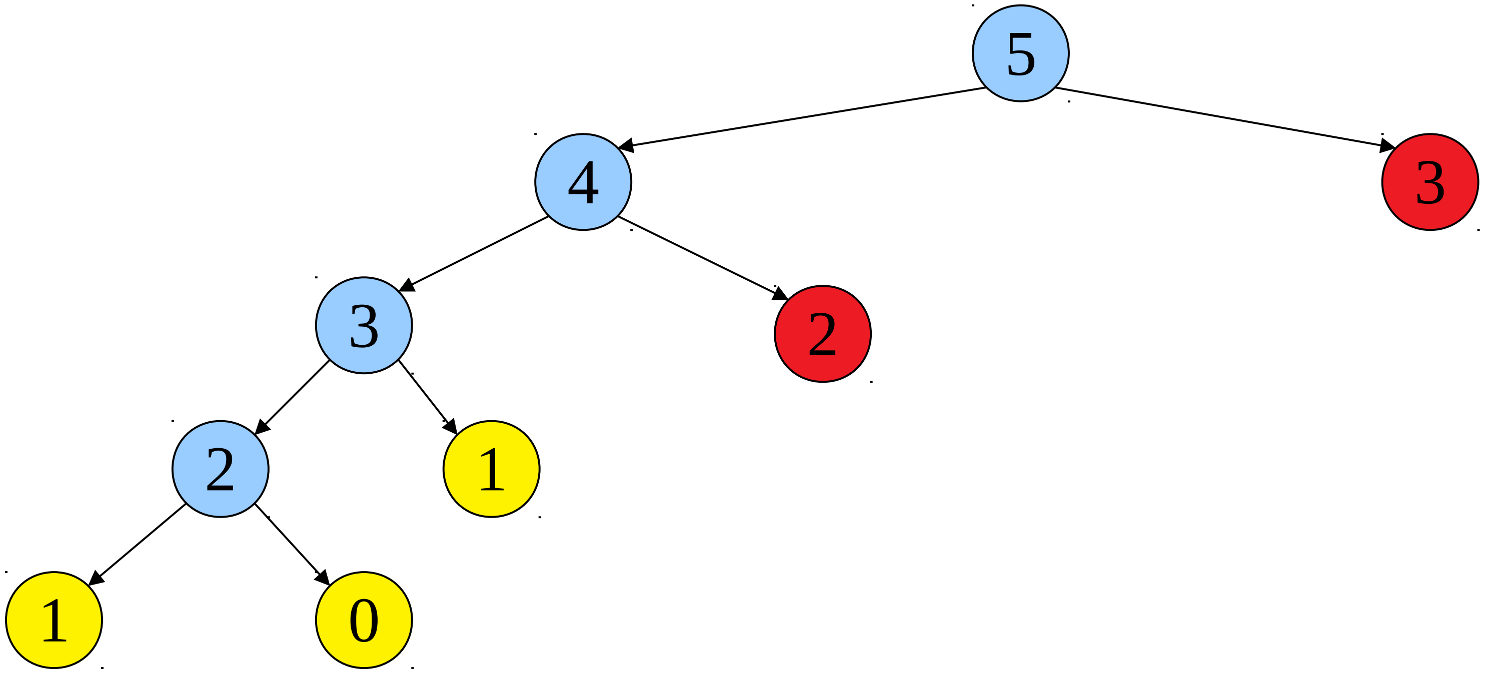


Programação Dinâmica

- Vamos estudar um esquema baseado em memoização
- Não é erro de grafia: vem de memo (memorando, anotação)
- Mantemos vetor com soluções já calculadas de subproblemas
  - F_2
  - F_3
  - F_4 ...

Programação Dinâmica

- Árvore de recursão



Programação Dinâmica

- Implementação com programação dinâmica:

```
#define MAX 46
int memo[MAX];
int inicializa() {
    for (int i = 2 ; i < MAX ; i++) memo[i] = -1;
    memo[0] = memo[1] = 1;
}
int fibonacci (int n) {
    if (memo[n] == -1) memo[n] = fibonacci(n-1)+fibonacci(n-2);
    return memo[n];
}
```

Programação Dinâmica

- Exemplo 2: Problema do troco
- Problema clássico da Ciência da Computação
- Suponha que você trabalhe em um mercado, e que o troco deve ser entregue ao cliente minimizando o total de cédulas/moedas entregues
- Por exemplo, troco de R$28, com estoque infinito de cédulas/moedas M={0.1, 0.25, 0.5, 1, 2, 5, 10, 20, 50, 100}
- Grande número de combinações: {20, 5, 2, 1} (4 cédulas/moedas), {10, 10, 5, 2, 1} (5 cédulas/moedas), {5, 5, 5, 5, 5, 2, 1} (7 cédulas/moedas), ...

Programação Dinâmica

- Implementação com programação dinâmica:

```
#define MAX_M 7
#define MAX_V 100
#define minimo(a,b) a<b?a:b
int M[MAX_M] = {1, 2, 5, 10, 20, 50, 100};
int memo[MAX_M][MAX_V];
void inicializa(){
   for (int i =0 ; i < MAX_M ; i++)
     for (int j = 0 ; j < MAX_V ; j++)
       if (j == 0 ) memo [i][j] = 0;
       else memo[i][j] = INT_MAX/2;
}
```

Programação Dinâmica

- Implementação com programação dinâmica:

```
int troco (int indice, int val) {
  if (val < 0 || indice == MAX_M) return INT_MAX/2;

  if (memo[indice][val] == INT_MAX/2){
    memo[indice][val] = minimo (1 + troco(indice,val-M[indice]),
                                      troco(indice+1,val));
  }
  return memo[indice][val];
}
```

Algoritmos Gulosos

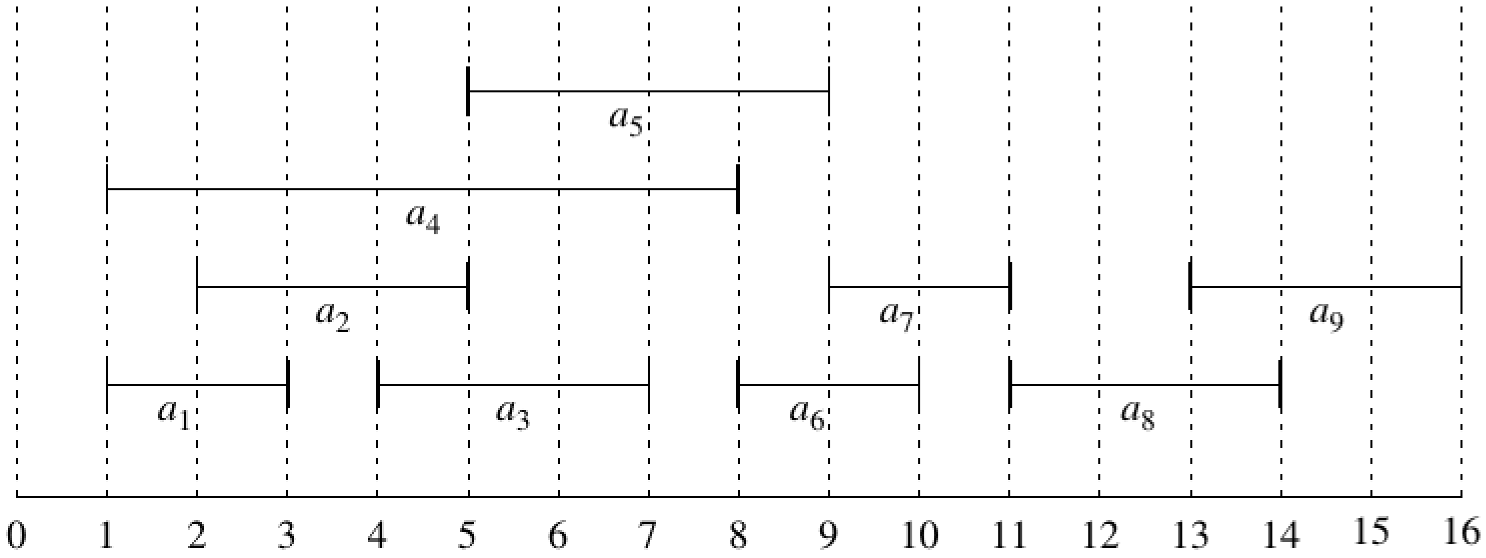
- Algoritmos para problemas de otimização
- Ideia: Quando temos escolhas a fazer, fazemos a melhor escolha do momento: escolha ótima local
- Algoritmos gulosos nem sempre geram uma solução ótima

Algoritmos Gulosos

- Problema da seleção de atividades
  - n atividades que requerem uso exclusivo de um recurso comum
  - Exemplo: escalonamento de uma sala de aula, escalonamento de um processador, etc;
  - Conjunto de atividades $S = \{a_1, \ldots, a_n\}$
  - Considere que a_i precisa do recurso durante o período $[s_i, f_i)$, onde s_i = horário de início e f_i = horário de término
  - Objetivo: selecionar o maior número possível de atividades que não se sobreponham no tempo
  - Assuma que atividades estejam ordenadas pelo horário de término: $f_1 <= f_2 <= \ldots <= f_n$

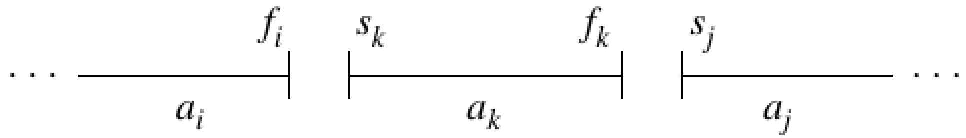
Algoritmos Gulosos

i	1	2	3	4	5	6	7	8	9
s_i	1	2	4	1	5	8	9	11	13
f_i	3	5	7	8	9	10	11	14	16



Algoritmos Gulosos

- Subestrutura ótima para seleção das atividades
  - $S_{ij} = \{a_k \in S : f_i <= s_k < f_k <= s_j\}$
  - Ou seja, atividades que começam depois que a_i termina, e terminam antes que a_j comece.



  - Seja A_{ij} o conjunto de tamanho máximo de atividades mutualmente compatíveis em S_{ij}

Algoritmos Gulosos

- Seja $a_k \in A_{ij}$ uma atividade em A_{ij}. Assim temos dois subproblemas
  - Encontrar atividades mutualmente compatíveis em S_{ik} (atividades que começam depois que a_i termina e que terminam depois que a_k começa)
  - Encontrar atividades mutualmente compatíveis em S_{kj} (atividades que começam depois que ak termina e que terminam depois que a_j começa)
  - Seja $A_{ik} = A_{ij} \cap S_{ik}$ = atividades em A_{ij} que terminam antes que a_k comeće
  - Seja $A_{kj} = A_{ij} \cap S_{kj}$ = atividades em A_{ij} que começam depois que a_k termine
  - Então $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj} \Rightarrow |A_{ij}| = |A_{ik}| + |A_{kj}| + 1$
  - Solução ótima para A_{ij} deve incluir soluções ótimas para os dois subproblemas S_{ik} e S_{kj}.

Algoritmos Gulosos

- Abordagem gulosa para resolver o problema
  - Encontrar uma atividade para adicionar a solução ótima antes de resolver os subproblemas
  - Para o problema de seleção de atividades, podemos escolher a atividade que deixa sala disponível para a maior quantidade de atividades possíveis
  - Ou seja, escolhemos a atividade que termina primeiro
  - Caso duas ou mais terminem juntas, podemos escolher qualquer uma delas
  - Uma vez que atividades ordenadas por tempo de término, escolhemos inicialmente a primeira, a_1
  - Tal escolha nos deixa com apenas um subproblema a ser resolvido: encontrar o maior conjunto de atividades mutualmente compatíveis que se iniciem após a_1 terminar

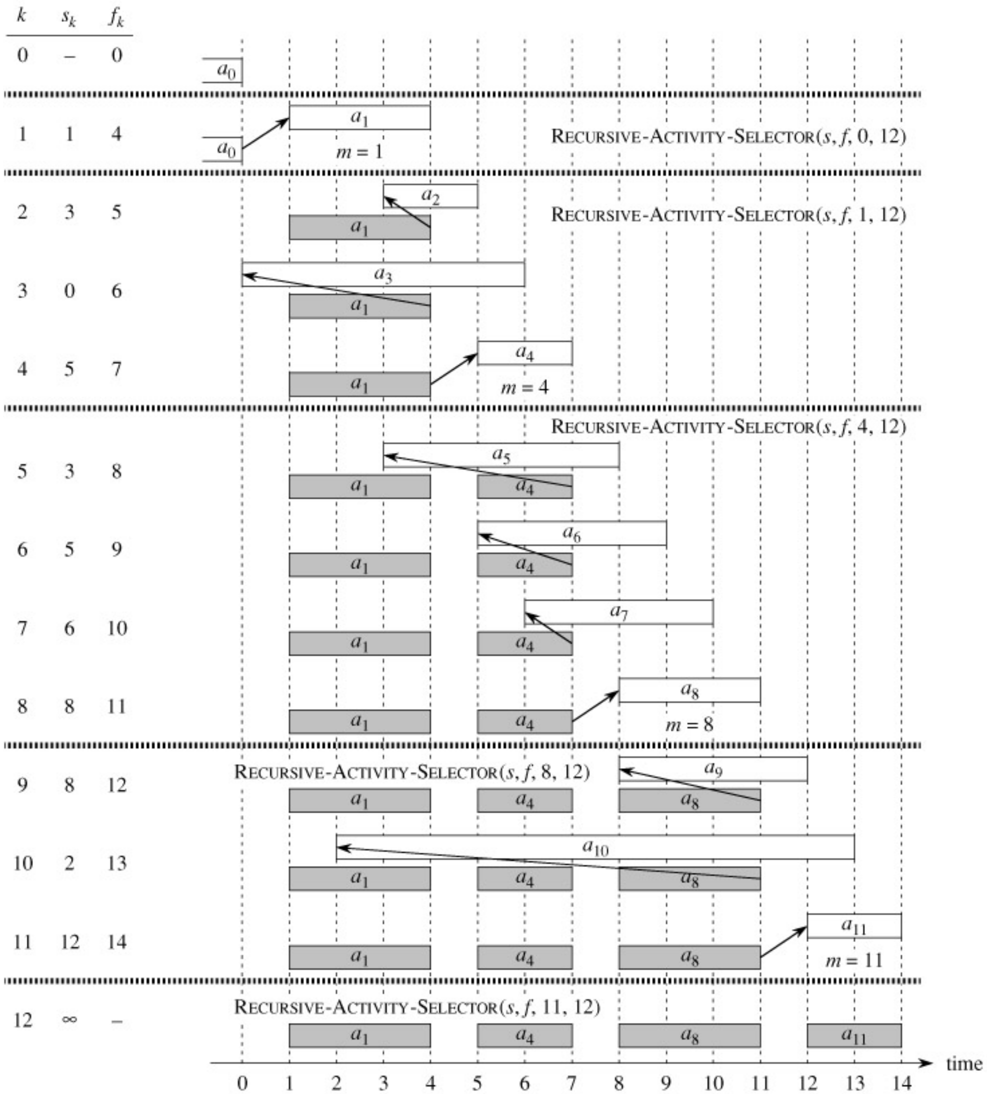
Algoritmos Gulosos

- Abordagem gulosa para resolver o problema
  - Podemos assim simplificar nossa notação: $S_k = \{a_i \in S : s_i >= f_k\}$ = atividades que iniciam após a_k terminar
  - Tempos de início e de término são representados por arranjos s e f, onde assume-se que f ordenado em ordem crescente
  - Para iniciar, adicionamos atividade fictícia a_0, com $f_0 = 0$

REC-ACTIVITY-SELECTOR(s, f, k, n)

```
m = k + 1
while m ≤ n and s[m] < f[k]          // find the first activity in S_k to finish
    m = m + 1
if m ≤ n
    return {a_m} ∪ REC-ACTIVITY-SELECTOR(s, f, m, n)
else return Ø
```

Algoritmos Gulosos



Próxima Aula...

- Classes de Problemas