Laboratórios de Algoritmia II

Programação dinâmica

Quando se aplica?

- Subestrutura óptima
  - A solução para um problema pode ser calculada à custa da solução dos subproblemas
  - Essencialmente o problema pode ser resolvido recursivamente
- Sobreposição de subproblemas
  - Na execução recursiva o mesmo subproblema é calculado muitas vezes

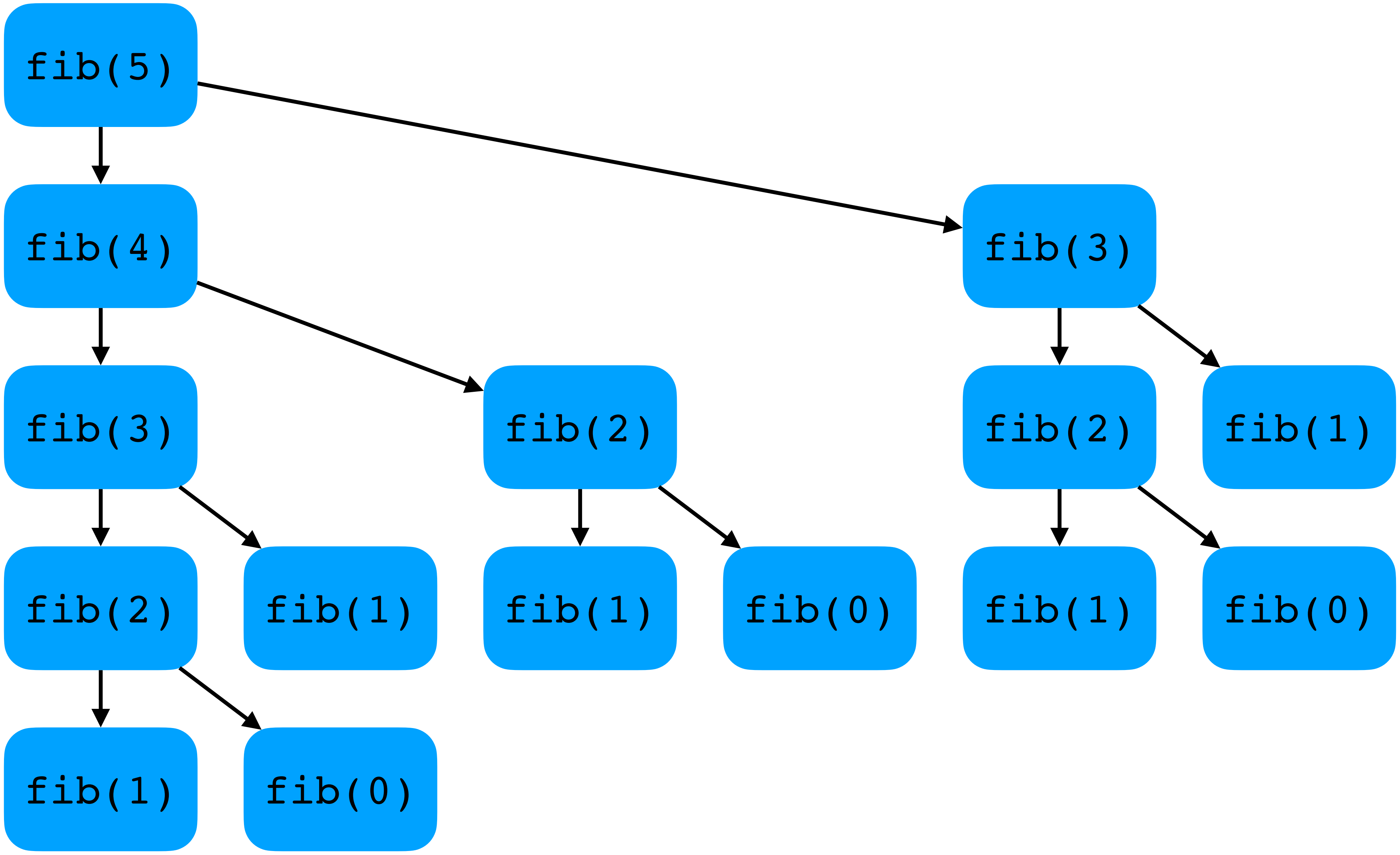
Como se optimiza?

- Top-down
  - Utiliza directamente a definição recursiva do problema
  - Usa-se *memoization* para armazenar soluções já calculadas num dicionário
  - Antes de calcular verifica-se no dicionário se a solução já está calculada e, nesse caso, devolve-se directamente o resultado
- Bottom-up
  - Inverte-se o calculo: tenta-se resolver primeiro os sub-problemas por forma a que as soluções já estejam disponíveis quando se vai resolver um problema maior
  - Frequentemente permite ter uma implementação iterativa em vez da formulação recursiva original
  - Esta estratégia é a normalmente conhecida como *programação dinâmica*

Fibonacci

```
def fib(n):
    if n<2:
        return 1
    else:
        return fib(n-1)+fib(n-2)
```

Fibonacci


fib(5)
fib(4)
fib(3)
fib(3)
fib(2)
fib(2)
fib(1)
fib(2)
fib(1)
fib(1)
fib(0)
fib(1)
fib(0)
fib(1)
fib(0)

Fibonacci

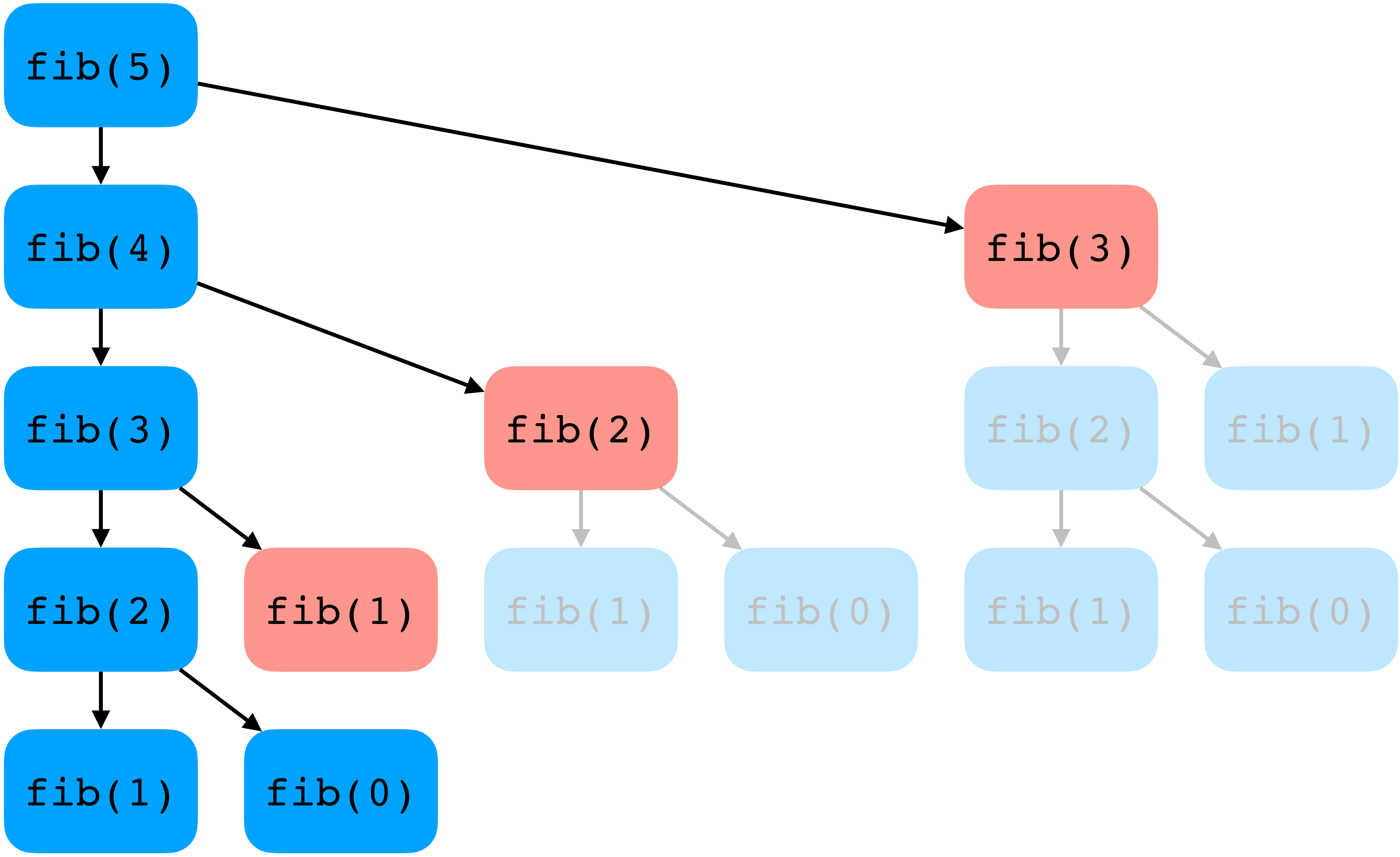
- Definição naturalmente (bi-)recursiva
- Definição muito ineficiente: tempo $O(2^n)$, espaço $O(n)$
- Não pode ser usada na prática
- Muitas invocações repetidas com o mesmo input
- Oportunidade para optimização com programação dinâmica

Fibonacci com *memoization*

```
def fib(n):
    return aux(n,{})

def aux(n,m):
    if n in m:
        return m[n]
    if n<2:
        m[n] = 1
    else:
        m[n] = aux(n-1,m)+aux(n-2,m)
    return m[n]
```

Fibonacci com *memoization*



Top-down vs *bottom-up*

- A definição com *memoization* já é muito mais eficiente: tempo O(n), espaço O(n)

- Para optimizar mais temos que usar a estratégia *bottom-up*

- Calculando `fib(0)`, depois `fib(1)`, depois `fib(2)`, …

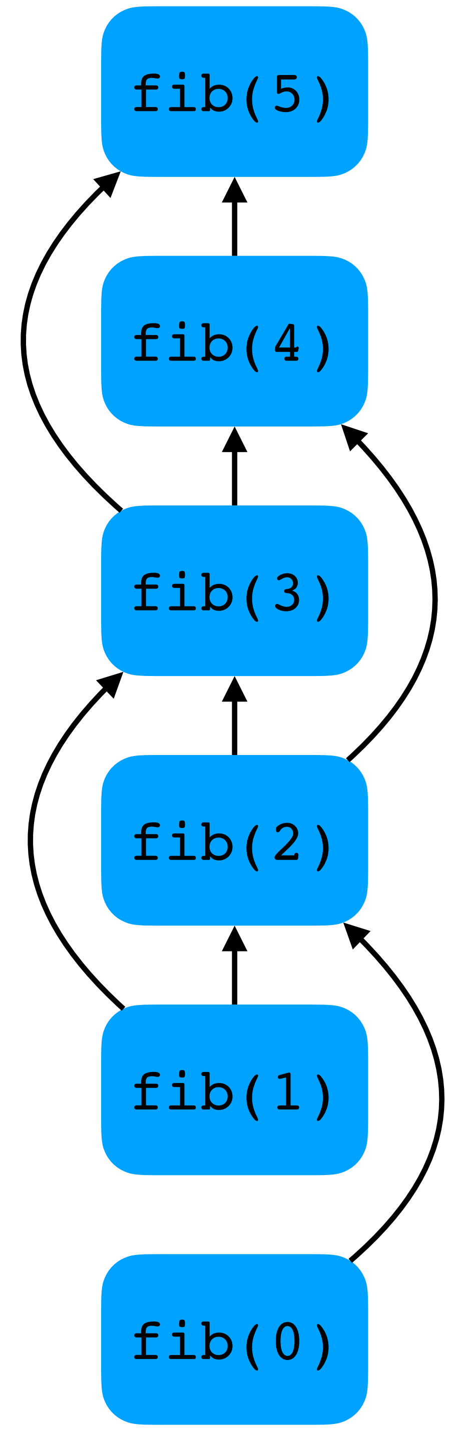
- Assim garante-se que já estão no dicionário todos os resultados "recursivos" quando se vai calcular o próximo número da série

- E podemos usar iteração em vez de recursividade, evitando-se também erros de *stack overflow* ou *maximum recursion depth exceeded*

Fibonacci com *programação dinâmica*

```
def fib(n):
    m = {}
    m[0] = 1
    m[1] = 1
    for i in range(2,n+1):
        m[i] = m[i-1] + m[i-2]
    return m[n]
```

Fibonacci com *programação dinâmica*



Fibonacci com *programação dinâmica*

- Definição ainda mais eficiente: tempo O(n), espaço O(n), mas constantes menores

- É possível optimizar mais?

- Não é necessário guardar no dicionário todos os valores previamente calculados

- Para calcular `fib(n)` apenas são necessários os 2 últimos resultados

- Se se apagar os resultados não necessários reduz-se a complexidade em termos de espaço para O(1)

- E nem é necessário usar um dicionário, bastam 2 variáveis inteiras

Fibonacci com *programação dinâmica*

```
def fib(n):
    m = {}
    m[0] = 1
    m[1] = 1
    for i in range(2,n+1):
        m[i] = m[i-1] + m[i-2]
        del m[i-2]
    return m[n]
```

Fibonacci com *programação dinâmica*

```
def fib(n):
    a = 1
    b = 1
    for i in range(2,n+1):
        # b == m[i-1] and a == m[i-2]
        b, a = a + b, b
    return b
```

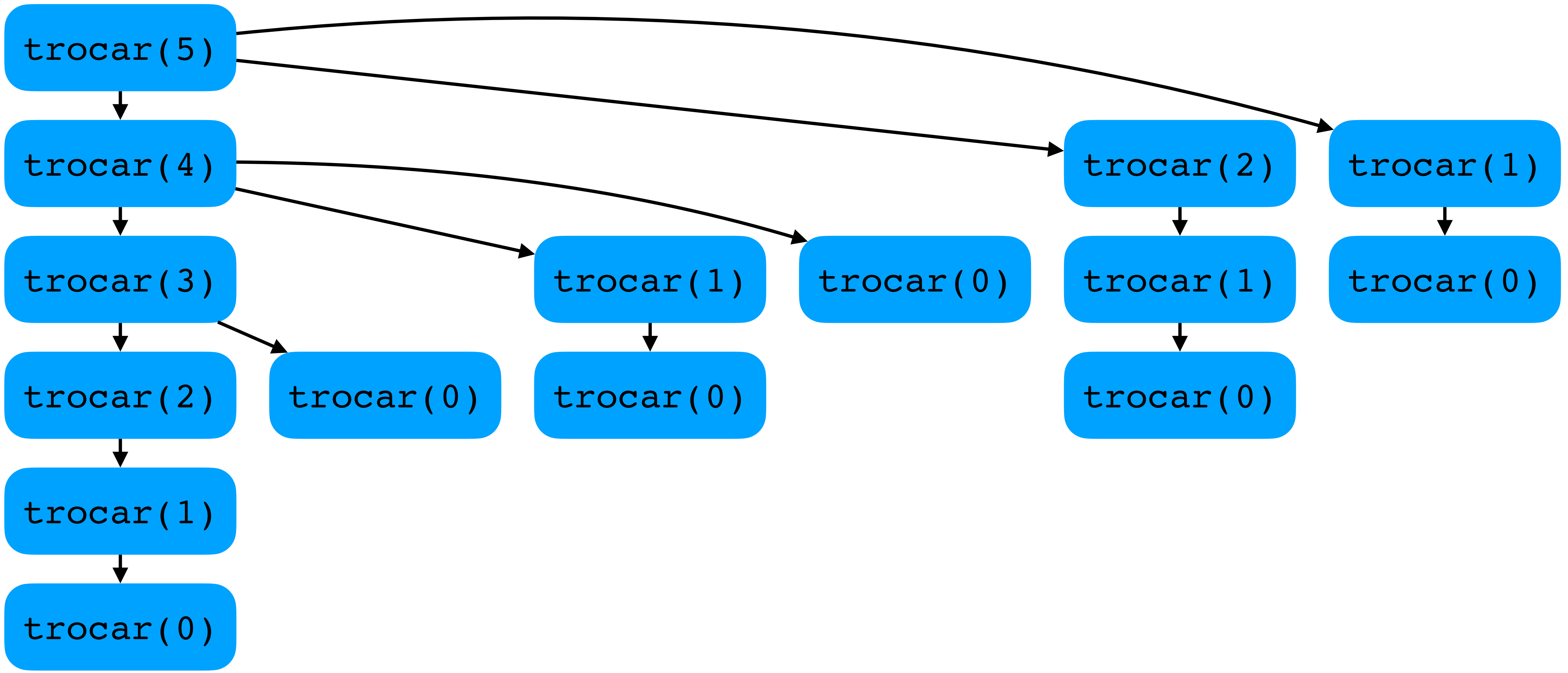
Trocar

- Como trocar uma quantia em moedas, minimizando o número de moedas?
- Assumindo um stock infinito de cada denominação
- Com sistemas de moedas canónicos pode-se adoptar uma solução *greedy*
- Se o sistema não for canónico esta estratégia não funciona (por exemplo trocar 6 com moedas de 1, 3 e 4)
- É possível formular uma solução recursiva elegante (mas ineficiente) para procurar a melhor solução

Trocar

```
def trocar(valor,moedas):
    if valor == 0:
        return 0
    r = float("inf")
    for m in moedas:
        if m <= valor:
            r = min(r,1+trocar(valor-m,moedas))
   return r
```

Trocar



trocar(5)
trocar(4)
trocar(2)
trocar(1)
trocar(3)
trocar(1)
trocar(0)
trocar(1)
trocar(0)
trocar(2)
trocar(0)
trocar(0)
trocar(0)
trocar(1)
trocar(0)

Trocar com *memoization*

```
def trocar(valor,moedas):
    return aux(valor,moedas,{})

def aux(valor,moedas,d):
    if valor in d:
        return d[valor]
    if valor == 0:
        d[valor] = 0
        return 0
    r = float("inf")
    for m in moedas:
        if m <= valor:
            r = min(r,1+aux(valor-m,moedas,d))
    d[valor] = r
    return r
```

Trocar com *programação dinâmica*

```
def trocar(valor,moedas):
    d = {}
    d[0] = 0
    for v in range(1,valor+1):
        r = float("inf")
        for m in moedas:
            if m <= v:
                r = min(r,1+d[v-m])
        d[v] = r
        if v >= max(moedas):
            del d[v-max(moedas)]
    return d[valor]
```

Trocar

- Como determinar quais as moedas a devolver?
- Pode-se guardar a melhor escolha para cada valor

Trocar com *programação dinâmica*

```
def trocar(valor,moedas):
    d = {}
    c = {}
    d[0] = 0
    for v in range(1,valor+1):
        r = float("inf")
        for m in moedas:
            if m <= v:
                a = 1+d[v-m]
                if r > a:
                    r = a
                    c[v] = m
        d[v] = r
        if v >= max(moedas):
            del d[v-max(moedas)]
    t = {m:0 for m in moedas}
    while valor > 0:
        t[c[valor]] += 1
        valor -= c[valor]
    return t
```

Trocar com *programação dinâmica*

v	d[v]	c[v]
0	0	
1	1	1
2	2	1
3	1	3
4	1	4
5	2	1
6	2	3
7	2	3
8	2	4
9	3	1
10	3	3

Distância de edição

- Qual o menor número de edições para transformar uma string noutra?
- Edições possíveis: remover um caracter, inserir um caracter, trocar um caracter por outro
- Por exemplo a distância entre `“invention”` e `“execution”` é 5
- `“invention” “nvention” “evention” ”exention” “exection” “execution”`
- Note que remoções e inserções são duais
- É possível formular uma solução recursiva elegante (mas ineficiente) para procurar a melhor solução

Distância de edição

```
def dist(a,b):
    if a == '':
        return len(b)
    elif b == '':
        return len(a)
    elif a[0] == b[0]:
        return dist(a[1:],b[1:])
    else:
        return min(1+dist(a,b[1:]),
                   1+dist(a[1:],b),
                   1+dist(a[1:],b[1:]))
```

Distância de edição com *memoization*

```
def dist(a,b):
    return aux(a,b,{})

def aux(a,b,m):
    if (a,b) in m:
        return m[(a,b)]
    if a == '':
        m[(a,b)] = len(b)
    elif b == '':
        m[(a,b)] = len(a)
    elif a[0] == b[0]:
        m[(a,b)] = aux(a[1:],b[1:],m)
    else:
        m[(a,b)] = min(1+aux(a,b[1:],m),
                       1+aux(a[1:],b,m),
                       1+aux(a[1:],b[1:],m))
    return m[(a,b)]
```

Distância de edição com *memoization*

- Usar pares de strings como chaves não é muito eficiente
- É possível usar apenas os índices do caracteres que estão a ser analisados em cada momento
- Também é indiferente analisar as strings da esquerda para a direita ou ao contrário

Distância de edição com *memoization*

```
def dist(a,b):
    return aux(a,len(a),b,len(b),{})

def aux(a,i,b,j,m):
    if (i,j) in m:
        return m[(i,j)]
    if i == 0:
        m[(i,j)] = j
    elif j == 0:
        m[(i,j)] = i
    elif a[i-1] == b[j-1]:
        m[(i,j)] = aux(a,i-1,b,j-1,m)
    else:
        m[(i,j)] = min(1+aux(a,i,b,j-1,m),
                       1+aux(a,i-1,b,j,m),
                       1+aux(a,i-1,b,j-1,m))
    return m[(i,j)]
```

Distância de edição com *programação dinâmica*

```
def dist(a,b):
    m = {}
    for j in range(len(b)+1):
        m[(0,j)] = j
    for i in range(len(a)+1):
        m[(i,0)] = i
    for j in range(1,len(b)+1):
        for i in range(1,len(a)+1):
            if a[i-1] == b[j-1]:
                m[(i,j)] = m[(i-1,j-1)]
            else:
                m[(i,j)] = min(1+m[(i,j-1)],
                               1+m[(i-1,j)],
                               1+m[(i-1,j-1)])
        for i in range(0,len(a)+1):
            del m[(i,j-1)]
    return m[(len(a),len(b))]
```

Distância de edição com *programação dinâmica*

		e	x	e	c	u	t	i	o	n
	0	1	2	3	4	5	6	7	8	9
i	1	1	2	3	4	5	6	6	7	8
n	2	2	2	3	4	5	6	7	7	7
v	3	3	3	3	4	5	6	7	8	8
e	4	3	4	3	4	5	6	7	8	9
n	5	4	4	4	4	5	6	7	8	8
t	6	5	5	5	5	5	5	6	7	8
i	7	6	6	6	6	6	6	5	6	7
o	8	7	7	7	7	7	7	6	5	6
n	9	8	8	8	8	8	8	7	6	5