Recursion & Dynamic Programming

Algorithm Design & Software Engineering
March 17, 2016
Stefan Feuerriegel

Today's Lecture

Objectives

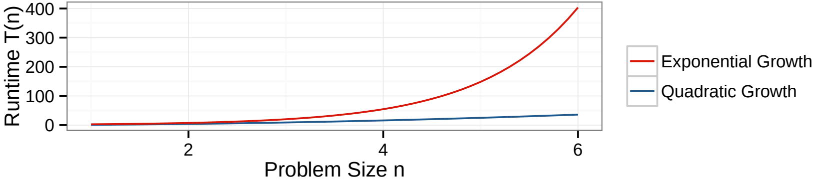
1. Specifying the complexity of algorithms with the big O notation
2. Understanding the principles of recursion and divide & conquer
3. Learning the contrary concept of dynamic programming

Outline

Outline

Need for Efficient Algorithms

- Computers can perform billions of arithmetic operations per second, but this might not be sufficient
- Memory is also limited
- Need for efficient algorithms that scale well



Examples

- Go games last up to 400 moves, with around 250 choices per move
- Cracking a 2048 bit RSA key theoretically requires 2^{112} trials
- Calculating the optimal order of delivering n parcels has $n!$ possibilities

Computational Complexity

- Computational complexity is measured by time $T(n)$ and space $S(n)$
- Exact measurements (e. g. timings) have shortcomings
  - Dependent on specific hardware setup
  - Difficult to convert into timings for other architectures
  - Cannot describe how well the algorithm scales
  - Initialization times are usually neglected

Approach

- Count operations as a function of problem size n
- Analyze best-case, average and worst-case behavior

Computational Complexity

Question

- What is the default number of operations for $\|\boldsymbol{x}\|_2 = \sqrt{\boldsymbol{x}^T\boldsymbol{x}}$ for $\boldsymbol{x} \in \mathbb{R}^n$?
  - 1 square root, 1 multiplication
  - 1 square root, n multiplications and $n-1$ additions
  - 1 square root, n multiplications and n additions
- No Pingo available

Question

- What better upper bound can one achieve for summing over n numbers?
  - $n-1$ additions
  - $\lfloor n/2 \rfloor$ additions
  - $\log n$ additions
- No Pingo available

Big O Notation

- Big O notation (or Landau O) describes the asymptotic or limiting behavior
- What happens when input parameters become very, very large
- Groups functions with the same growth rate (or an upper bound of that)
- Common functions

Notation	**Name**	**Example**
$O(1)$	Constant	Test if number is odd/even
$O(\log n)$	Logarithmic	Finding an item in a sorted list
$O(n \log n)$	Loglinear	Sorting n numbers
$O(n)$	Linear	Dot product, finding an item in an unsorted list
$O(n^2)$	Quadratic	Default matrix-by-vector multiplication
$O(n^c)$	Polynomial	Default matrix-by-matrix multiplication
$O(c^n)$	Exponential	Traveling salesman problem with DP

Big O Notation

Example: testing if a number n is prime can be in $O(n)$ or $O(\sqrt{n})$

1. Variant in $O(n)$
   Iterate all numbers in the range $i = 2, \ldots, n$ and check if i is an integer divisor of n

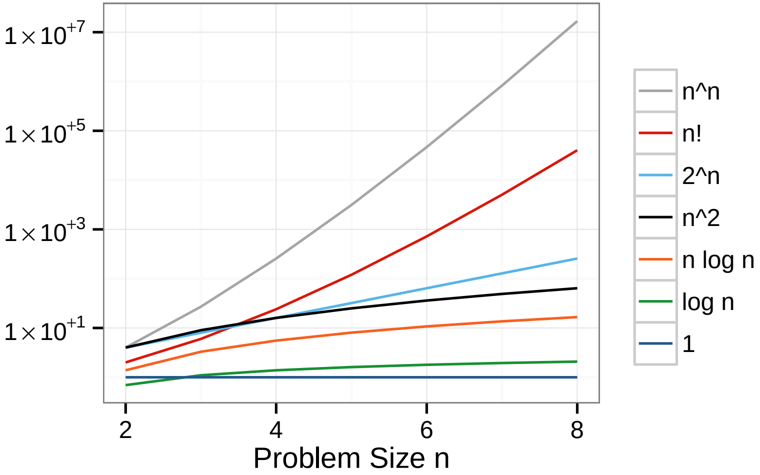
```
for (i in 2:n) {
  if (n%%i == 0) {
    print("Not prime")
  }
}
```

2. Variant in $O(\sqrt{n})$
   Changing for-loop as the largest possible divisor is $\sqrt{n}$

```
for (i in 2:sqrt(n)) {
  if (n%%i == 0) {
    print("Not prime")
  }
}
```

Complexity Classes

Even small cases are intractable for inefficient algorithms



Big O Notation

- Classifies algorithms by how they respond (e. g. in their runtime) to changes in input size
- Gives the worst-case complexity regarding time or space
- Let $f, g : \mathbb{N} \mapsto \mathbb{R}^+$, we define

  $$f(x) = O(g(x)) \qquad \text{as } x \to \infty$$

  if and only if

  $$|f(x)| \leq c\,|g(x)| \qquad \forall x \geq x_0$$

  where c is a positive constant
- Mathematically correct is $f(x) \in O(g(x))$, though more common is $f = O(g)$ where "$=$" means "is"

Big O Notation

Example

- Assume $f(x) = 5x^2 + 10x + 20$
- $5x^2$ is the highest growth rate
- To prove $f(x) = O(g(x))$ with $g(x) = x^2$, let $x_0 = 1$ and $c = 35$

$$
\begin{aligned}
& & |f(x)| &\leq c|g(x)| \\
\Leftrightarrow & & |5x^2 + 10x + 20| &\leq 5x^2 + 10x + 20 \\
\Rightarrow & & |5x^2 + 10x + 20| &\leq 5x^2 + 10x^2 + 20x^2 \\
\Leftrightarrow & & |5x^2 + 10x + 20| &\leq 35x^2 \\
\Leftrightarrow & & |10x + 20| &\leq 30x^2
\end{aligned}
$$

- Hence $f(x) = 5x^2 + 10x + 20 = O(x^2)$

O Calculus

Multiplication by a constant

- Let $c \neq 0$ be a constant, then $O(c\,g) = O(g)$
  $\rightarrow$ e. g.changing the underlying hardware does not affect the complexity
- Example: $5 \cdot O(1) = O(1)$

Sum rule

- Let $f_1 = O(g_1)$ and $f_2 = O(g_2)$, then $f_1 + f_2 = O(g_1 + g_2)$
  $\rightarrow$ e. g. complexity of sequentially executing two algorithms is only affected by the one with the higher complexity
- As a special case: $f_1 = O(g)$ and $f_2 = O(g) \Rightarrow f_1 + f_2 = O(g)$
- Example: $O(x) + O(x^2) = O(x^2)$

Product rule

- Assume $f_1 = O(g_1)$ and $f_2 = O(g_2)$, then $f_1\,f_2 = O(g_1\,g_2)$
  $\rightarrow$ e. g. matches nested execution of loops

Big O Notation

- The complexity class is a set of functions defined by

$$O(g) = \{f : \mathbb{N} \mapsto \mathbb{R}^+ \mid \exists x_0 \in N, c \in \mathbb{R}, c > 0 \; \forall x \geq x_0 : f(x) \leq c\, g(x)\}$$

- Alternative to prove $f = O(g)$ is to show that the following limits exists

$$\lim_{x \to \infty} \frac{f(x)}{g(x)} < \infty$$

Example

- Show $f(x) = 5x^2 + 10x + 20 = O(x^2)$
- Apply limit theory

$$\lim_{x \to \infty} \frac{f(x)}{x^2} = \lim_{x \to \infty} \frac{5x^2 + 10x + 20}{x^2} = 5 < \infty$$

Related Notations

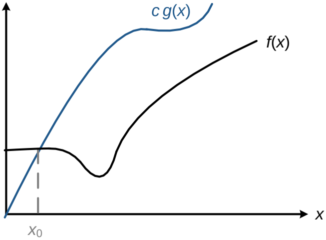
Different variants exists for all forms of lower and upper bounds, e. g.

1. Upper bound $f = O(g)$: f grows not faster than g
2. Lower bound $f = \Omega(g)$: f grows at least as quickly as g
3. Tight bound $f = \Theta(g)$: f and g grow at the same rate

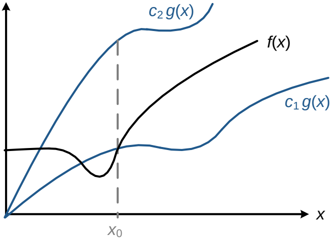
If $f = \Omega(g)$ and $f = O(g)$, then $f = \Theta(g)$ because

$$\underbrace{c_1 f(x)}_{=\Omega(g)} \leq \underbrace{f(x)}_{=\Theta(f)} \leq \underbrace{c_2 f(x)}_{=O(g)} \qquad \forall x \geq x_0$$

1



3



Outline

Recursion

Idea

- Design algorithm to solve a problem by progressively solving smaller instances of the same problem

Definition

Recursion is a process in which a function calls itself with:

1. a base case which terminates the recursion
   $\rightarrow$ Producing an answer without a recursive call
2. a set of rules which define how a base case is finally reached

Pseudocode

```
recursion <- function(...) {
  if (condition) {
    return(base_case)
  }
  return(recursion(...)) # recursive call
}
```

Factorial

- Factorial is the product of all positive numbers less or equal n
- Example: $5! = \prod_{i=1}^{5} i = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 = 120$
- Recurrence equation

$$n! := \begin{cases} 1 & \text{if } n = 1 \\ n \cdot (n-1)! & \text{if } n \geq 2 \end{cases}$$

- Number of function calls is $\Theta(n)$

```
fact  <- function(n)  {
 if  (n  ==  0)  {
   return(1)
 }
  return(n  *  fact(n-1))
}
fact(5)


##  [1]  120
```

Fibonacci Numbers

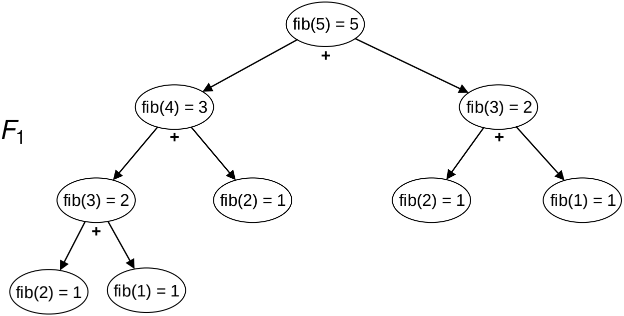
Fibonacci sequence of integers F_n for $n \in \mathbb{N}$

- Recurrence equation $F_n := F_{n-1} + F_{n-2}$ for $n \geq 3$
- Base cases $F_1 := 1$ and $F_2 := 1$

Example

$$
\begin{aligned}
& F_5 \\
= & F_4 + F_3 \\
= & \underbrace{F_3 + F_2}_{=F_4} + \underbrace{F_2 + F_1}_{=F_3} \\
= & \underbrace{F_2 + F_1}_{=F_3} + F_2 + F_2 + F_1 \\
= & 1 + 1 + 1 + 1 + 1 \\
= & 5
\end{aligned}
$$

Call stack for $n = 5$



Fibonacci Numbers

Implementation

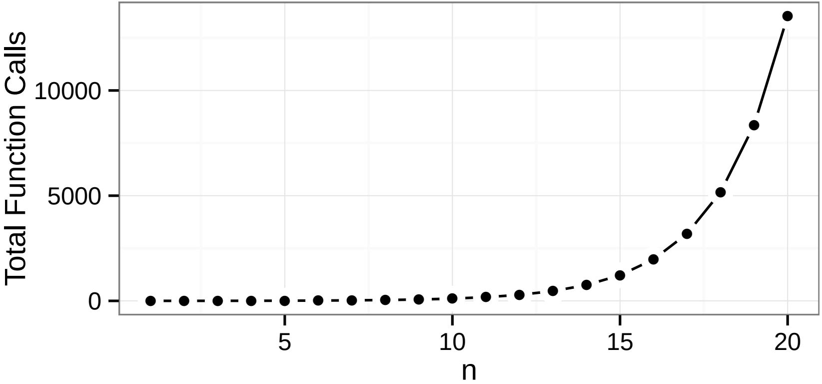
```r
fibonacci  <- function(n) {
  # base case
  if (n == 1 || n == 2) {
    return(1)
  }

  # recursive calls
  return(fibonacci(n-1) + fibonacci(n-2))
}
```

Example

```r
fibonacci(5)

## [1] 5
```

Computational Complexity

Superlinear growth in the number of function calls

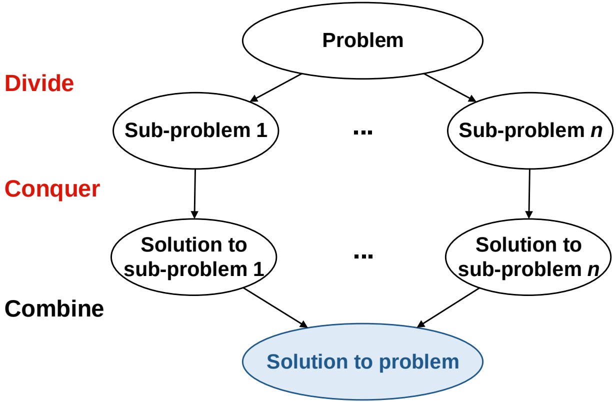


- Number of function calls is in $O(F_n) = O(c^n)$
- But there are more efficient approaches, such as dynamic programming or a closed-form formula

Divide and Conquer

Divide and conquer (D&C) is an algorithm pattern based on recursion

1. **Divide** complex problem into smaller, non-overlapping sub-problems
2. **Conquer**, i. e. find optimal solution to these sub-problems recursively
3. Combine solutions to solve the bigger problem



Common tasks

- Sorting
- Computational geometry
- MapReduce in Hadoop

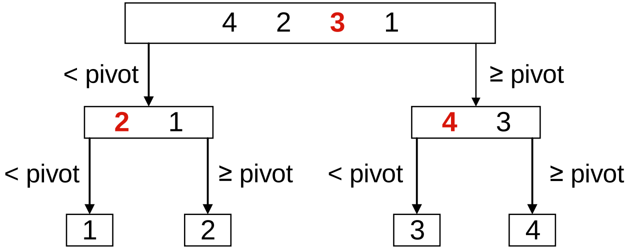
Quicksort

- Quicksort is an efficient sorting algorithm for sorting $\boldsymbol{x} \in \mathbb{R}^n$
- Average performance is $O(n \log n)$, worst-case $O(n^2)$

Idea

1. Choose pivot element from vector $\boldsymbol{x}$
2. Partitioning: split into two parts $x_i <$ pivot and $x_i \geq$ pivot
3. Recursively apply to each part (and combine)

Example



Quicksort

- Function `quicksort` based on divide and conquer

```
quicksort <- function(x) {
  # base case
  if (length(x) <= 1) {
    return(x)
  }

  # pick random pivot element for "divide"
  pivot <- x[sample(length(x), 1)]

  # recursive "conquer"
  return(c(quicksort(x[x < pivot]),     # "left"
           quicksort(x[x >= pivot])))   # "right"
}
```

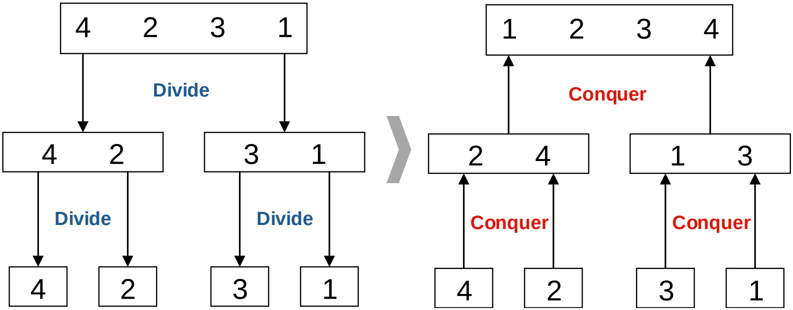
- Example

```
quicksort(c(4, 2, 3, 1))

## [1] 1 2 3 4
```

Mergesort

Idea

1. Divide vector into sub-vectors until you have vectors of length 1
2. Conquer by merging sub-vectors recursively (i. e. fill in right order)



Average and worst-case performance are in $O(n \log n)$ but with higher factor in practice

Mergesort

1. **Divide** operation to split vector and then merge

```
mergesort <- function(x) {
  if (length(x) == 1) {
    return(x)
  }

  # divide recursively
  split <- ceiling(length(x)/2)
  u <- mergesort(x[1:split])
  v <- mergesort(x[(split+1):length(x)])

  # call conquer step
  return(merge(u, v))
}
```

Mergesort

2. **Conquer** operation which merges two vectors recursively

```
merge <- function(u, v) { # input u and v must be sorted
  # new empty vector
  x.sorted <- numeric(length(u) + length(v))
  # indices pointing to element processed next
  uj <- 1
  vj <- 1

  for (i in 1:length(x.sorted)) {
    # case 1: v is completely processed => take u
    # case 2: still elements in u and the next
    #         u element is smaller than the one in v
    if (vj > length(v) ||
        (uj <= length(u) && u[uj] < v[vj])) {
      x.sorted[i] <- u[uj]
      uj <- uj + 1
    } else { # Otherwise: take v
      x.sorted[i] <- v[vj]
      vj <- vj + 1
    }
  }
  return(x.sorted) # return sorted vector
}
```

Mergesort

- Example of merging two vectors as part of conquer step

```
merge(4, 2)

## [1] 2 4

merge(c(2, 4), c(1, 3))

## [1] 1 2 3 4
```

- Example of mergesort

```
mergesort(4)

## [1] 4

mergesort(c(4, 2))

## [1] 2 4

mergesort(c(4, 2, 3, 1))

## [1] 1 2 3 4
```

Outline

Dynamic Programming

Idea

- Dynamic programming (DP) uses a memory-based data structure
- Reuse previous results
- Speed-ups computation but at the cost of increasing memory space

Approach

- Divide a complex problem into smaller, overlapping sub-problems
- Find optimal solution to these sub-problems and remember them
- Combine solutions to solve the bigger problem
- All sub-problems are ideally solved exactly once

Implementation

- Sub-problems can be processed in top-down or bottom-up order
- Bottom-up DP is usually faster by a constant factor

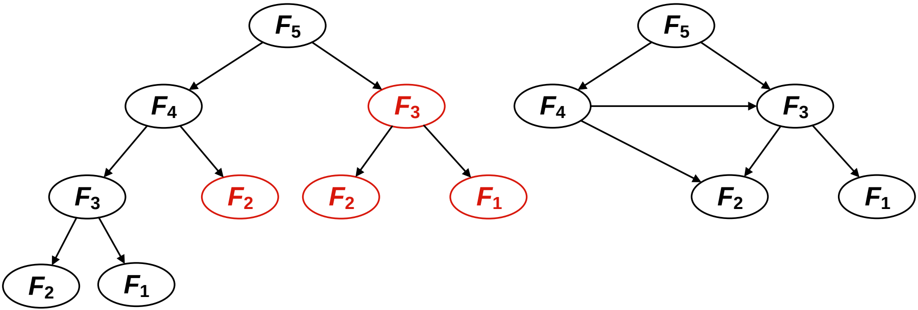
Dynamic Programming

Example: Fibonacci numbers

- Recursive approach triggers many redundant function calls
- Reordering these function calls can improve the performance

Dependencies in recursion

Dependencies in DP



Dynamic Programming

Top-down approach: memoization

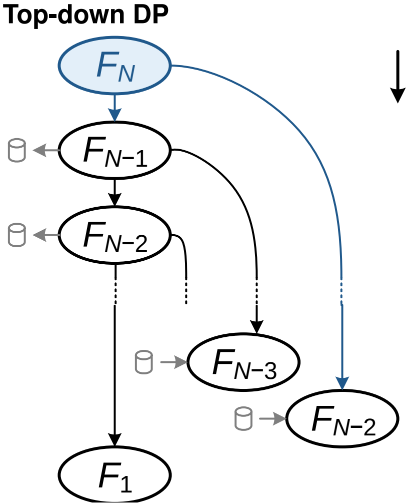
- Store solutions to sub-problems in a table that is build top-down
- For each sub-problem, check table first if there is already a solution
  1. If so, reuse it
  2. If not, solve sub-problem and add solution to table

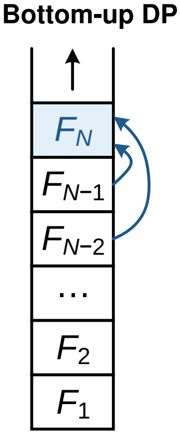
Bottom-up approach: tabulation

- Store solutions to sub-problems in a table that is build bottom-up
- Start with the smallest sub-problem
- Solve sub-problem and store solution
- Use solutions of smaller sub-problems to solve bigger ones

Comparison

Example: Fibonacci numbers with top-down dynamic programming





Top-Down DP

```r
fib_topdown <- function(n) {
  # check cache first
  if (!is.na(memo_table[n])) {
    return(memo_table[n])
  }

  # solve sub-problem
  if (n == 1 || n == 2) {
    F <- 1
  } else {
    F <- fib_topdown(n-1) + fib_topdown(n-2)
  }

  memo_table[n] <<- F  # add solution to global table
  return(F)
}
```

Operator <<− sets a global variable outside the function

Top-Down DP

- Set up example $n = 5$

```
n <- 5
```

- Create global memoization table

```
memo_table <- rep(NA, n)
```

- Run top-down algorithm

```
fib_topdown(n)

## [1] 5
```

- View memoization table after execution

```
memo_table

## [1] 1 1 2 3 5
```

Bottom-Up DP

Example: Fibonacci numbers with bottom-up dynamic programming

```r
fib_bottomup <- function(n) {
  # initialization of table
  fib <- numeric(n)
  fib[1] <- 1
  fib[2] <- 1

  # bottom-up construction
  for (i in 3:n) {
    fib[i] <- fib[i-1] + fib[i-2]
  }

  return(fib[n])
}

fib_bottomup(5)

## [1] 5
```

Comparison: Fibonacci Numbers

Recursion

- Runtime $O(c^n)$ becomes infeasible even for small n
- Direct space is $O(1)$ but the call stack needs $O(n)$

Top-down DP

- Runtime $O(n)$
- Space $O(n)$ and additional call stack in $O(n)$

Bottom-up DP

- Runtime $O(n)$
- Space $O(n)$ and no call stack
  $\rightarrow$ Storing only last two values can reduce it to $O(1)$

Knapsack Problem

Description

- Suppose you want to steal items from a shop
- Each item i has weight w_i and price p_i
- Which items to choose?

Optimization Problem

- Decision variable $x_i \in \{0, 1\}$ tells whether to include item i
- Your aim is to maximize your profit

$$P^* = \max_{x_1,\ldots,x_n} \sum_{i=1}^{n} x_i \, w_i$$

- Limitation is the capacity W of your bag

$$\sum_{i=1}^{n} x_i \, w_i \leq W$$

Knapsack Problem

Example

- Load necessary library

```r
library(adagio)
```

- Sample values

```r
w <- c(2, 4, 5,  7,  9)
p <- c(1, 5, 4, 10, 20)
W <- 13
```

- Solve problem with dynamic programming

```r
knapsack <- knapsack(w, p, W)
knapsack$profit # maximum possible profit

## [1] 25
```

- Print decision variable, i. e. which items to choose

```r
knapsack$indices

## [1] 2 5
```

DP for Knapsack Problem

- Build up matrix $M_{i,j}$ for $i = 0, \ldots, n$ and $j = 0, \ldots, W$
- $M_{i,j}$ is the maximum profit with items $1, \ldots, i$ and weight of up to j
- Solution to knapsack problem is given by $M_{n,W}$
- Recursive definition of M

$$M_{i,j} := \begin{cases} 0 & \text{if } i = 0 \\ M_{i-1,j} & \text{if } w_i > j \\ \max\left\{M_{i-1,j}, M_{i-1,j-w_i} + p_i\right\} & \text{if } w_i \leq j \end{cases}$$

- The reasons behind the cases are
  1. If $i = 0$, the set of items is empty
  2. If $w_i > j$, the new item exceeds the current weight limit j
  3. Choose between
     - Current-best solution without item i
     - Combining price p_i from item i and the recursive solution for the remaining, available weight $j - w_i$

Knapsack Problem

Bottom-up DP needs runtime $O(nW)$ and space $O(nW)$

```
n <- length(p) # total number of items
M <- matrix(0, nrow=n+1, ncol=W+1)

# note: matrix indexing starts with 1 in R
for (i in 1:n) {
  for (j in 0:W) {
    if (w[i] > j) {
      M[i+1, j+1] <- M[i, j+1]
    } else {
      M[i+1, j+1] <- max(M[i, j+1],
                         M[i, j-w[i]+1] + p[i])
    }
  }
}

M[n+1, W+1]

## [1] 25
```

Bellman's Principle of Optimality

Theoretical basis why DP "works"

- We find an optimal solution by solving each sub-problem optimally
- "An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision"

  $\rightarrow$ Bellman (1957), Chap. 3

- Such a problem is said to have a optimal sub-structure
  $\rightarrow$ Greedy algorithms also produce good results
- Proof via backward induction
- Also highly relevant in game theory and reinforcement learning

Bellman's Principle of Optimality

Problem

- Let's assume an optimization problem with $t = 1, \ldots, T$ steps
- Payoff $F(t, a_t)$ in step t given decision a_t ("action")
- $V(t)$ is the optimal total utility when starting in step t
- Calculated by maximizing

$$V(1) = \max_{a_1, \ldots, a_T} \sum_{t=1}^{T} F(t, a_t)$$

Bellman equation

- Separate today's decision a_1 from future decisions $a_2, \ldots, a_T$
- We can rewrite $V(1)$ in a recursive fashion

$$V(1) = \max_{a_1} \left[F(1, a_1) + V(2) \right]$$

Outline

1. Computational Complexity
2. Recursion
3. Dynamic Programming
4. Greedy Algorithms

Greedy Algorithms

Idea

- Make locally optimal choices with the hope of finding global optimum
- Depending on problem, greedy algorithms can find optimum or only work as heuristic
- This depends on the specific setting $\rightarrow$ optimal sub-structure

Examples

- Decision tree learning
- Routing in networks
- Heuristics for complex optimization problems

Greedy Algorithms

Change-Making Problem

- What is the minimum number of coins when giving change?
- Assume that there is an infinite amount available
- Greedy strategy is to give out largest coins first and then continue in decreasing order

Example: given 47 cents

$47 - 20 = 27$ (*20*)

$27 - 20 = 7$ (*20*) (*20*)

Skip 10

$7 - 5 = 2$ (*20*) (*20*) (*5*)

$2 - 2 = 0$ (*20*) (*20*) (*5*) (*2*)

Greedy Algorithm for Change-Making Problem

```
min_coins_change <- function(cents) {
  # list of available coins
  coins <- c(200, 100, 50, 20, 10, 5, 2, 1)
  num <- 0

  while (cents > 0) {
    if (cents >= coins[1]) {
      num <- num + 1 # one more coins
      cents <- cents - coins[1] # decrease remaining value
    } else {
      # switch to next smaller coin
      coins <- coins[-1] # remove first entry in list
    }
  }

  return(num)
}

min_coins_change(47) # 2x 20ct,  1x 5ct, 1x 2ct

## [1] 4
```

Heuristics

Idea

- Find an approximate solution instead of the exact solution
- Usually used when direct method is not efficient enough
- Trade-off: optimality vs. runtime
- Common applications: logistics, routing, malware scanners

Heuristic for Knapsack Problem

Solve Knapsack problem by choosing items in the order of their price-to-weight ratio

- Greedy algorithm
  1. Sort items by $\frac{p_i}{w_i}$ in decreasing order
  2. Pick items in that order as long as capacity constraint remains fulfilled
- Runtime $O(n)$ and space $O(1)$
- When number of items is unbounded ($x_i \geq 0$), the heuristic is only worse by fixed factor

  $\rightarrow$ Heuristic always gives at least $P^*/2$

Heuristic for Knapsack Problem

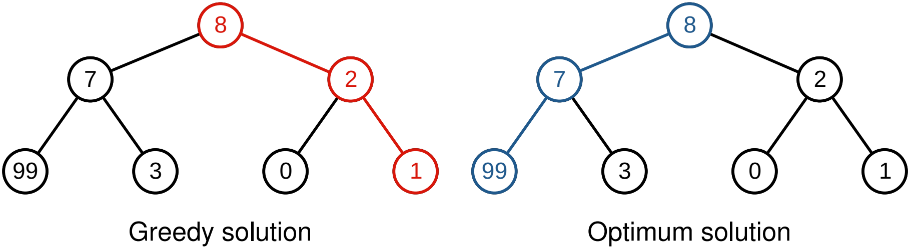
```
# generate greedy order according to price/weight ratio
greedy_order <- order(p/w, decreasing=TRUE)

sum_p <- 0 # total profit in current iteration
sum_w <- 0 # total weight in current iteration

for (i in greedy_order) {
  if (sum_w + w[i] <= W) {
    sum_p <- sum_p + p[i]
    sum_w <- sum_w + w[i]
  }
}

sum_p # optimal solution was 25

## [1] 25
```

Greedy Algorithms vs. Heuristics

- Greedy algorithms might fail to find the optimal solution
- This depends on the specific setting ($\rightarrow$ optimal sub-structure)
- Sometimes one can derive upper bounds

Example

- Task is to find the path with the highest total sum
- Greedy approach achieves 11, while optimum is 114



Summary

Computational complexity

- Even fast hardware needs efficient algorithms
- Algorithmic performance is measured by runtime and memory
- Asymptotic complexity is given by e. g. the big O notation (upper limit)

Recursion

- Progressively solves smaller instances of a problem
- Function calls itself multiple times until reaching a base case
- Specific pattern is the approach of divide and conquer

Summary

Dynamic programming

- Dynamic programming can be top-down or bottom-up
- Stores solutions of sub-problems and reuses them
- Decreases runtime at the cost of increasing memory usage

Greedy algorithms

- Making locally optimal choices to find or approximate global optimum
- Heuristics find approximate instead of exact solution
- Trade-off: solution quality vs. runtime