Rekursion

Koffman & Wolfgang

- kapitel 5

Rekursivt tänkande

Rekursion reducerar ett problem till en eller flera enklare versioner av samma problem.

- med ”enklare” menas att underproblemen måste vara ”mindre”, enligt något sätt att räkna
- t.ex. genom storleken på indata
- kallas också ”divide-and-conquer”

Stegen i en rekursiv algoritm

Det måste finnas ett eller flera fall som kan lösas direkt:

- basfall, för små värden på *n*

För alla större *n*-värden måste problemet kunna reduceras till en eller flera delproblem:

- rekursionsfall
- delproblemen är mindre versioner av samma problem
- alla delproblem måste vara mindre än huvudproblemet (mindre *n*-värden)
- det måste gå att kombinera delproblemens lösningar till en lösning av huvudproblemet

Enkla rekursiva definitioner

De enklaste rekursionerna är lite meningslösa:

```
public static int length(LinkedList<E> list) {
    if (list == null)
        return 0;
    else
        return 1 + length(list.next);
}
```

...för de kan lika gärna definieras med en loop:

```
public static int length(LinkedList<E> list) {
    int len = 0;
    while (list != null) {
        len++;
        list = list.next;
    }
    return len;
}
```

En till enkel rekursiv definition

Fakulteten är definierad som:

- $n! = n \cdot (n\text{-}1)!$, om $n>0$
- $0! = 1$

Den rekursiva definitionen är snarlik:

```
public static int factorial(int n) {
    if (n == 0)
        return 1;
    else
        return n * factorial(n-1);
}
```

...men den iterativa versionen är inte så komplicerad den heller:

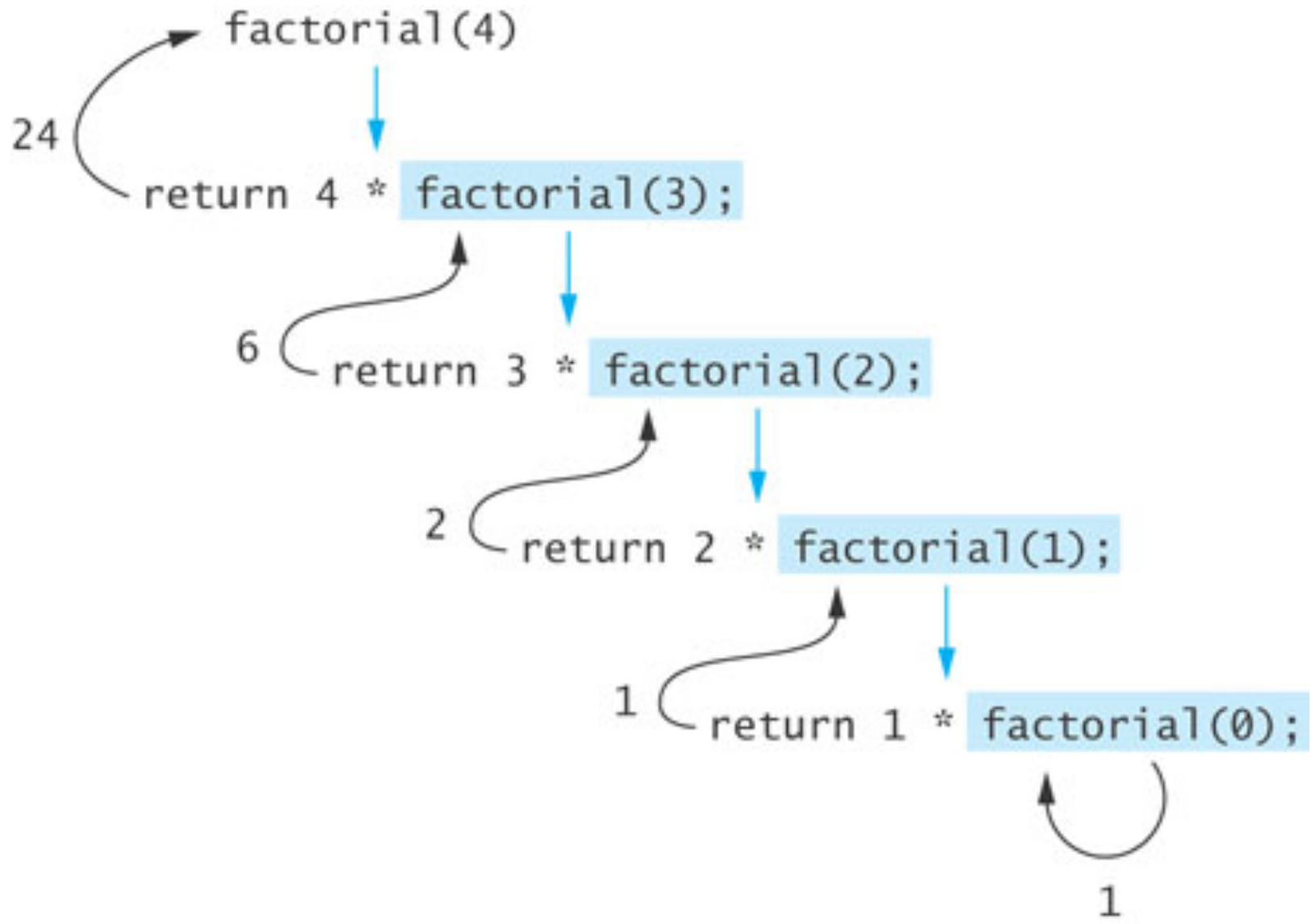
```
public static int factorial(int n) {
    int fac = 1;
    for (int i = 1; i <= n; i++)
        fac *= i;
    return fac;
}
```

Att bevisa korrekthet

Korrekthet för rekursiva algoritmer bevisas med induktion:

- visa att alla basfall täcks av implementationen, och löses korrekt
- visa att varje rekursivt anrop minskar n
- visa att om varje delproblem ($n' < n$) är löst korrekt, så blir också huvudproblemet (n) korrekt löst

Att "spåra" (trace) rekursion



Oändlig rekursion

Om du anropar *factorial* med ett negativt värde, så kommer rekursionen inte att terminera:

- till slut kommer du få ett `StackOverflowError`
- *factorial* saknar alltså ett basfall

Var alltid noga med att täcka alla möjliga basfall, även de felaktiga:

- t.ex. negativa värden
- om argumentet är felaktigt så kan du kasta ett `IllegalArgumentException`

Största gemensamma delaren

”Greatest common divisor” (gcd)

- gcd(*m*, *n*) är det största heltal som delar både *m* och *n*
- gcd(20, 15) = 5
- gcd(36, 24) = 12
- gcd(38, 18) = 2
- gcd(17, 97) = 1
  - dvs, 17 och 97 är relativt prima

Euklides algoritm för gcd

För att beräkna gcd(m, n), där $m \geq n$

- gcd(m, 0) = m
- gcd(m, n) = gcd(n, m % n)

Om $m < n$, så vänder vi bara på argumenten.

Euklides algoritm (forts.)

Rekursiv version:

```
public static long gcd(long m, long n) {
    if (n == 0)
        return m;
    else
        return gcd(n, m % n);
}
```

eller iterativ:

```
public static long gcd(long m, long n) {
    while (n != 0) {
        long temp = m % n;
        m = n;
        n = temp;
    }
    return m;
}
```

Euklides algoritm (forts.)

Rekursiv version:

```
public static long gcd(long m, long n) {
    if (n == 0)
        return m;
    else
        return gcd(n, m % n);
}
```

eller iterativ:

```
public static long gcd(long m, long n) {
    while (n != 0) {
        long temp = m % n;
        m = n;
        n = temp;
    }
    return m;
}
```

OBS!
detta förutsätter att
$m \geq n$

det är en ***invariant***,
som vi måste visa
alltid är sann

vilket den är eftersom
$n \geq (m \,\%\, n)$

Rekursion vs iteration

Rekursion och iteration är lika kraftfulla:

- det går alltid att översätta
  en rekursiv lösning till en iterativ
  - enklare rekursioner översätts till iterativa loopar
  - mer komplexa rekursioner kräver en eller flera stackar
- det går alltid att översätta
  en iterativ lösning till en rekursiv
  - loopar översätts till svansrekursiva funktioner

Vilket som är bäst är en fråga om tycke och smak

Fibonacciserien

Definition:

- $fib_0 = 0$
- $fib_1 = 1$
- $fib_n = fib_{n-1} + fib_{n-2}$

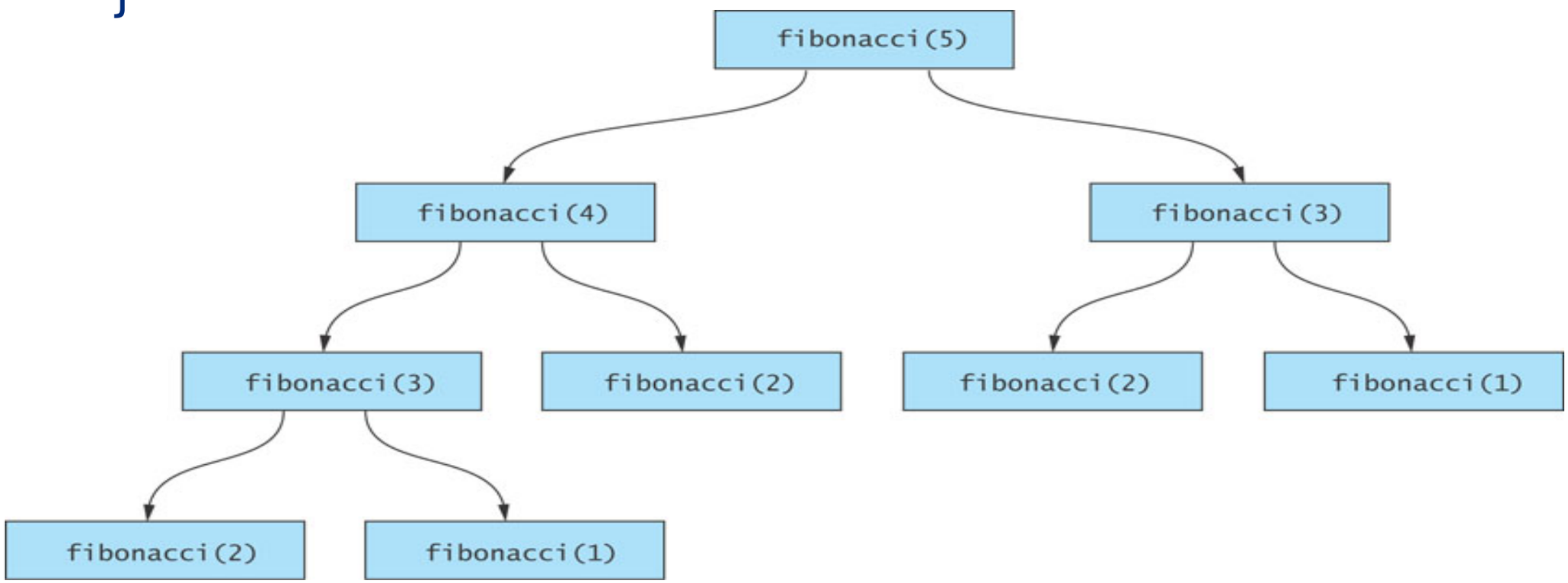
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...

En dum implementation

```
public static long fibonacci(long n) {
    if (n <= 1)
        return n;
    else
        return fibonacci(n - 1) + fibonacci(n - 2);
}
```

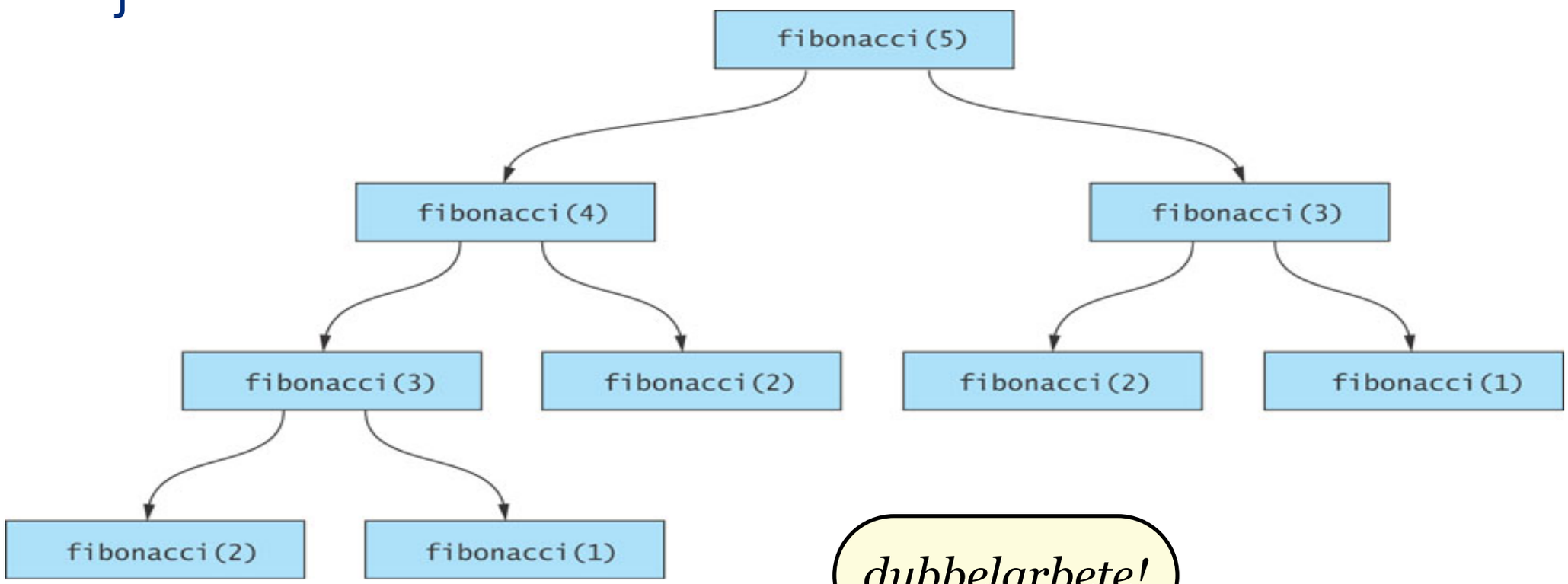
En dum implementation

```
public static long fibonacci(long n) {
    if (n <= 1)
        return n;
    else
        return fibonacci(n - 1) + fibonacci(n - 2);
}
```



En dum implementation

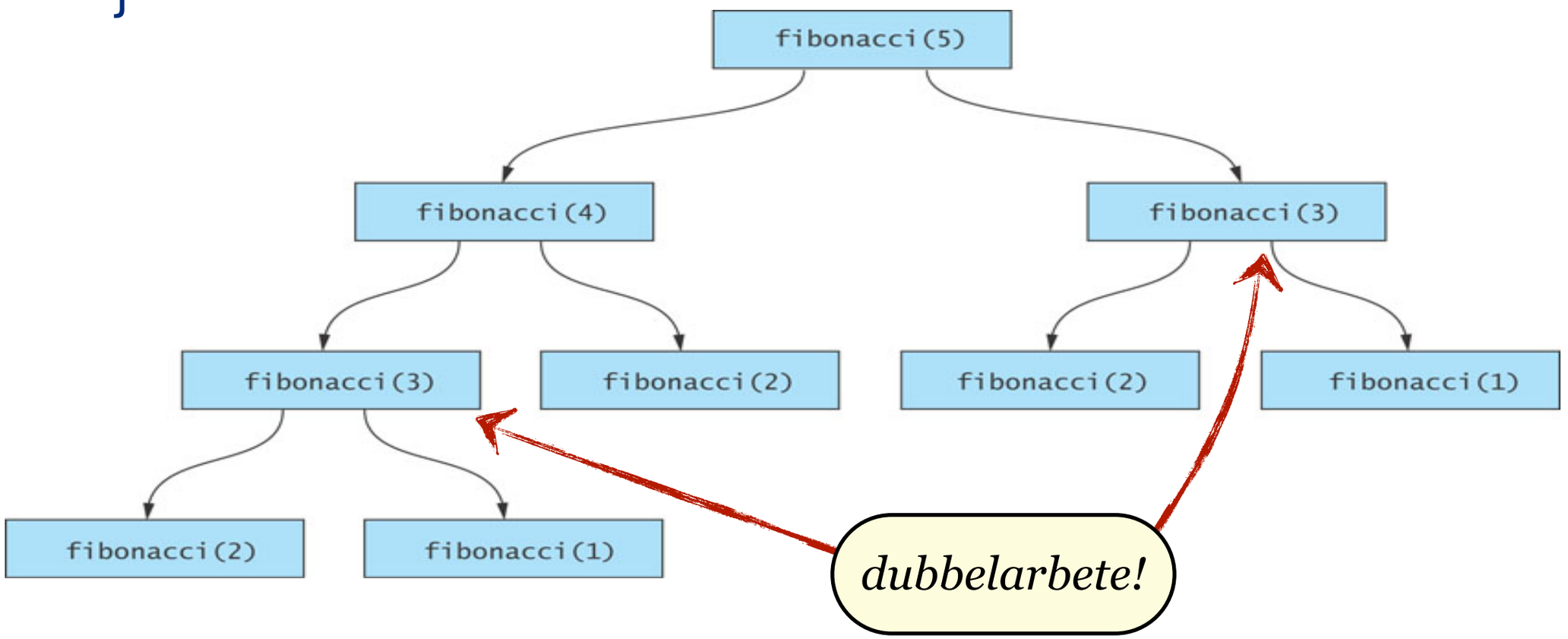
```
public static long fibonacci(long n) {
    if (n <= 1)
        return n;
    else
        return fibonacci(n - 1) + fibonacci(n - 2);
}
```



dubbelarbete!

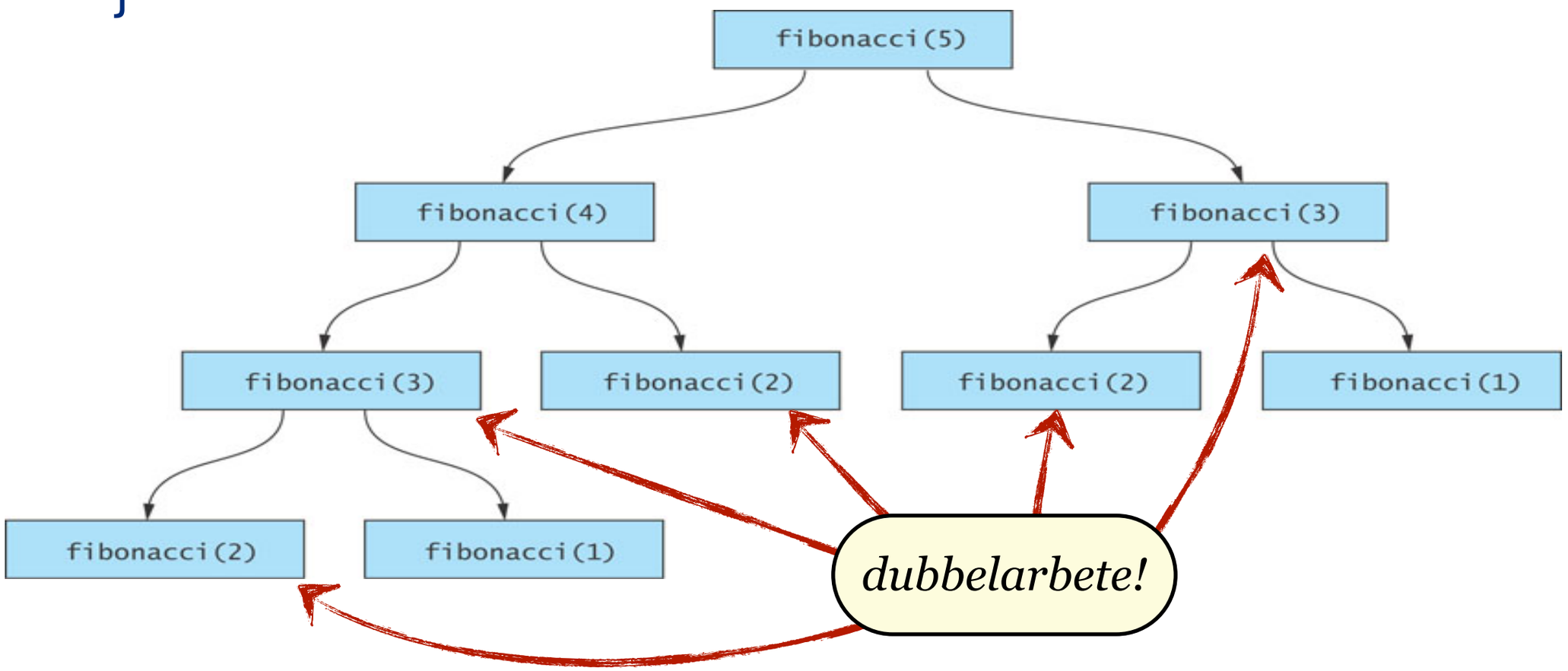
En dum implementation

```
public static long fibonacci(long n) {
    if (n <= 1)
        return n;
    else
        return fibonacci(n - 1) + fibonacci(n - 2);
}
```



En dum implementation

```
public static long fibonacci(long n) {
    if (n <= 1)
        return n;
    else
        return fibonacci(n - 1) + fibonacci(n - 2);
}
```



En bättre implementation

```
public static long fibonacci(long n) {
    return fibo(1, 0, n);
}

static long fibo(long current, long previous, int n) {
    if (n == 1)
        return current;
    else
        return fibo(current + previuos, current, n - 1);
}
```

fib_k

fib_{k-1}

fib_{k+1} = $fib_k + fib_{k-1}$

fib_k

En iterativ implementation

```
public static long fibonacci(long n) {
    long previous = 0;
    long current = 1;
    for (var i = 2; i <= n; i++) {
        long temp = previous;
        previous = current;
        current += temp;
    }
    return current;
}
```

Binärsökning i ett sorterat fält

Antag att vi ska leta efter ett objekt i ett *sorterat* fält:

- då kan vi jämföra vårt objekt med mittelementet
- om de inte är lika så fortsätter vi leta, antingen i elementen till vänster eller till höger om mittelementet
- istället för att fortsätta leta bland $n-1$ element, så letar vi bland $n/2$ element

Binärsökning (forts.)

Rekursiv algoritm för att söka efter ett objekt i ett fält:

- om fältet är tomt:
  - returnera –1
- annars, om målobjektet är lika med mittelementet:
  - returnera mittelements position
- annars, om målobjektet är mindre än mittelementet:
  - sök rekursivt bland elementen som ligger före mittelementet
- annars är målobjektet större än mittelementet:
  - sök rekursivt bland elementen som ligger efter mittelementet

Comparable-gränssnittet

Comparable-gränssnittet deklarerar metoden `compareTo()`

Metodanropet obj_1.`compareTo`(obj_2) ska returnera ett heltal med följande värden:

- *negativt* om $obj_1 < obj_2$
- *noll* om $obj_1 == obj_2$
- *positivt* om $obj_1 > obj_2$

Binärsökning i ett fält

```
public static int binarySearch(Object[] items, Comparable target) {
    return binarySearch(items, target, 0, items.length - 1);
}


private static int binarySearch(Object[] items, Comparable target,
                                int first, int last) {
    if (first > last)
        return -1;      // Base case for unsuccessful search.
    else {
        int middle = (first + last) / 2;  // Next probe index.
        int compResult = target.compareTo(items[middle]);
        if (compResult == 0)
            return middle;   // Base case for successful search.
        else if (compResult < 0)
            return binarySearch(items, target, first, middle - 1);
        else
            return binarySearch(items, target, middle + 1, last);
    }
}
```

Binärsökning, exempel

Caryn	Debbie	Dustin	Elliot	Jacquie	Jonathon	Rich
0	1	2	3	4	5	6

Binärsökning, exempel

target = Dustin

Caryn	Debbie	Dustin	Elliot	Jacquie	Jonathon	Rich
0	1	2	3	4	5	6

Binärsökning, exempel

target = Dustin

Första anropet

Caryn	Debbie	Dustin	Elliot	Jacquie	Jonathon	Rich
0	1	2	3	4	5	6

Binärsökning, exempel

target = Dustin

Första anropet

Caryn	Debbie	Dustin	Elliot	Jacquie	Jonathon	Rich
0	1	2	3	4	5	6

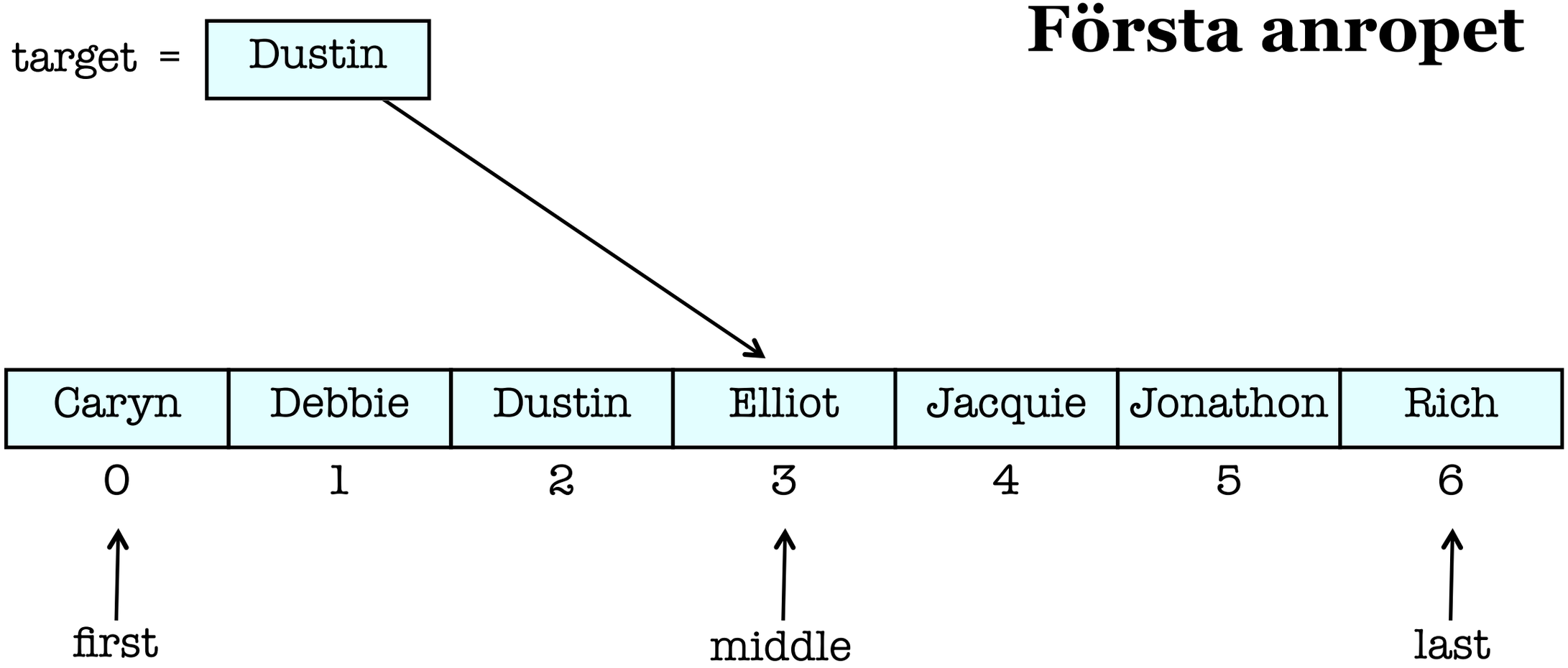
first

middle

last

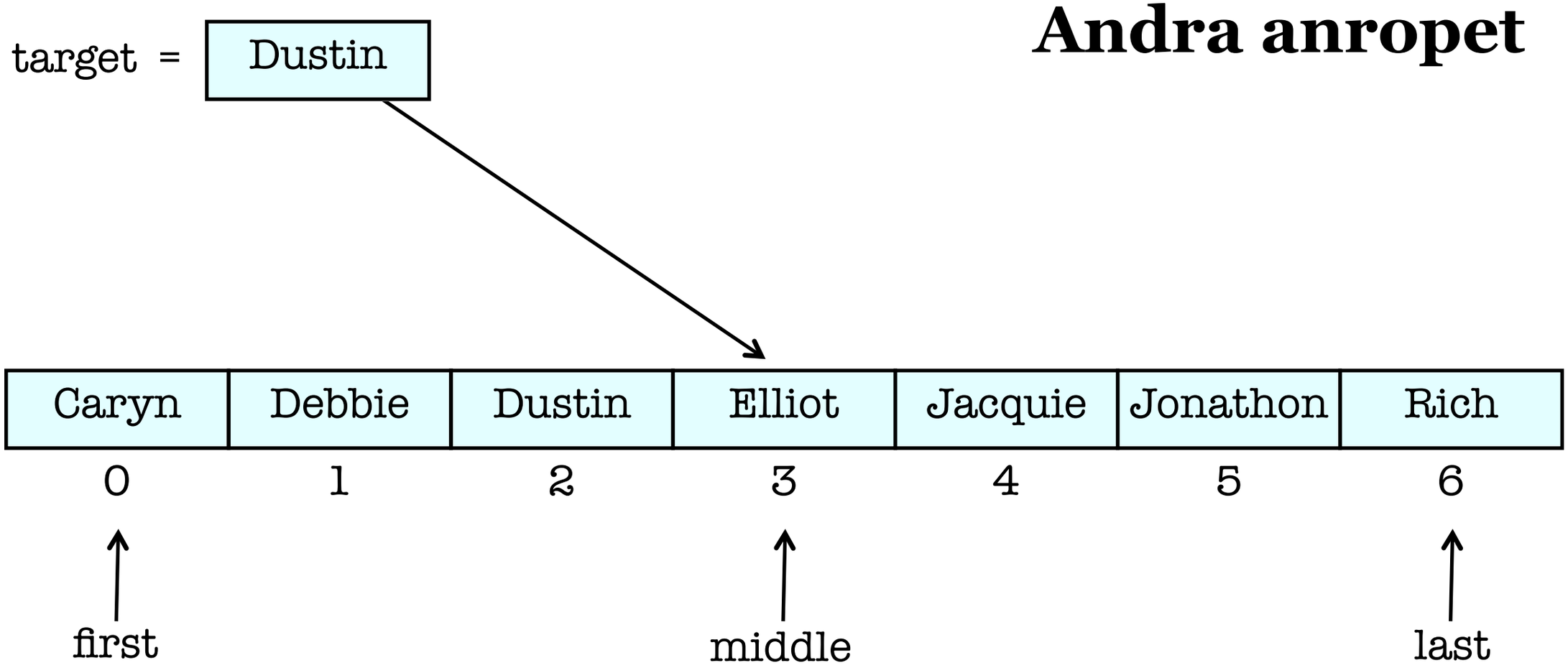
Binärsökning, exempel

Första anropet



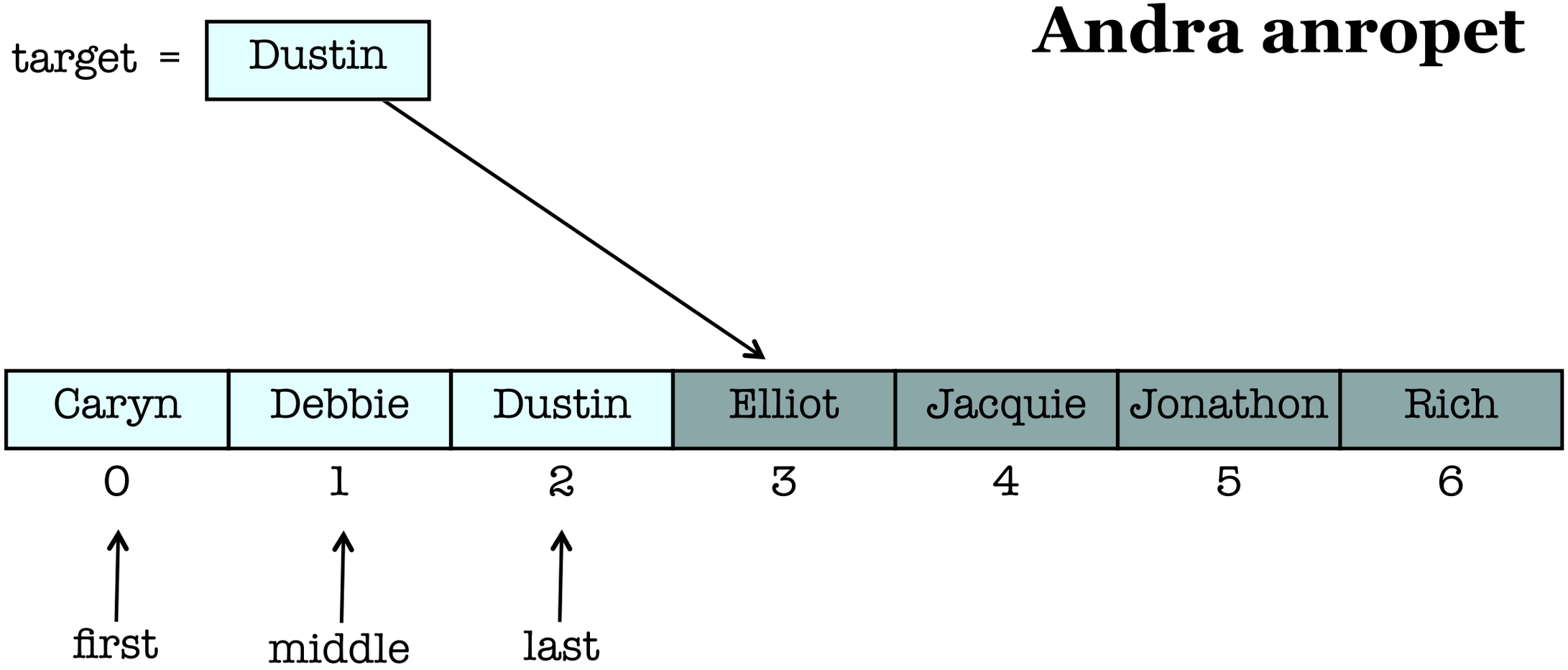
Binärsökning, exempel

Andra anropet



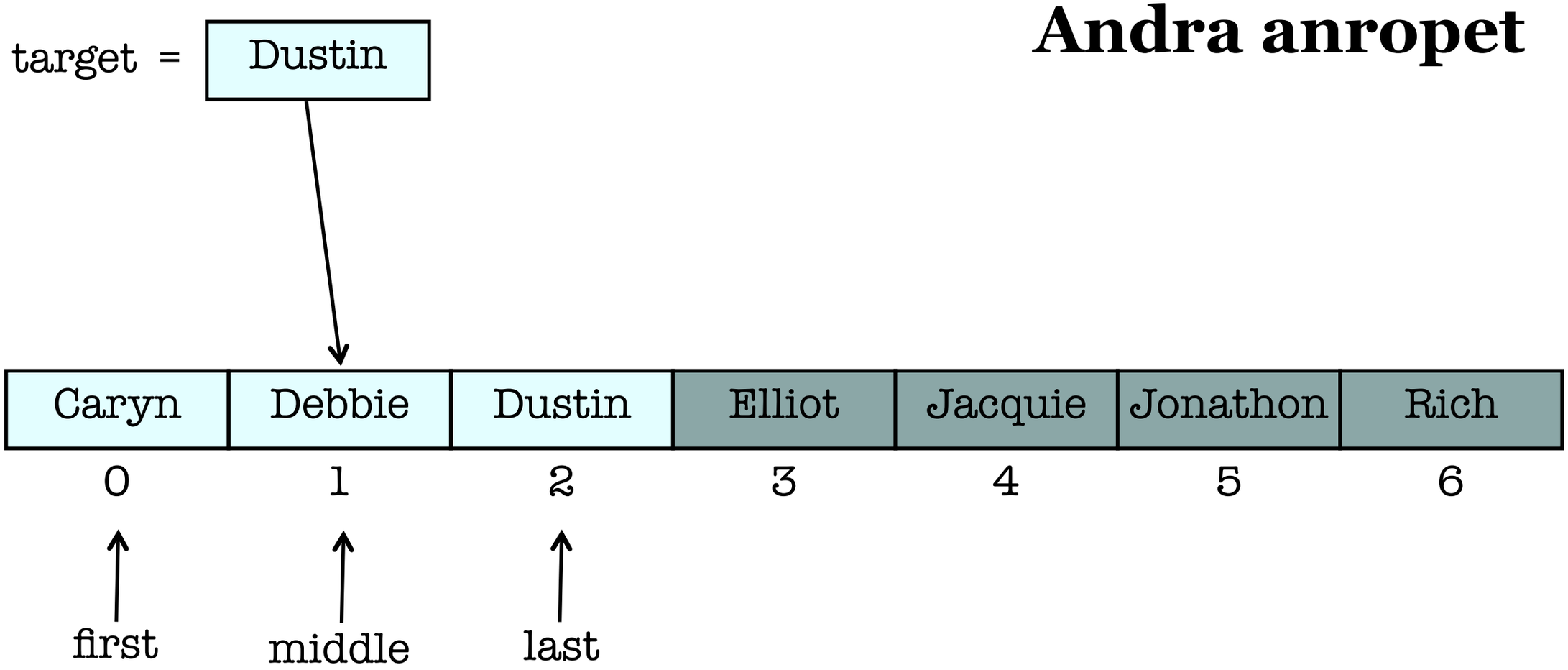
Binärsökning, exempel

Andra anropet



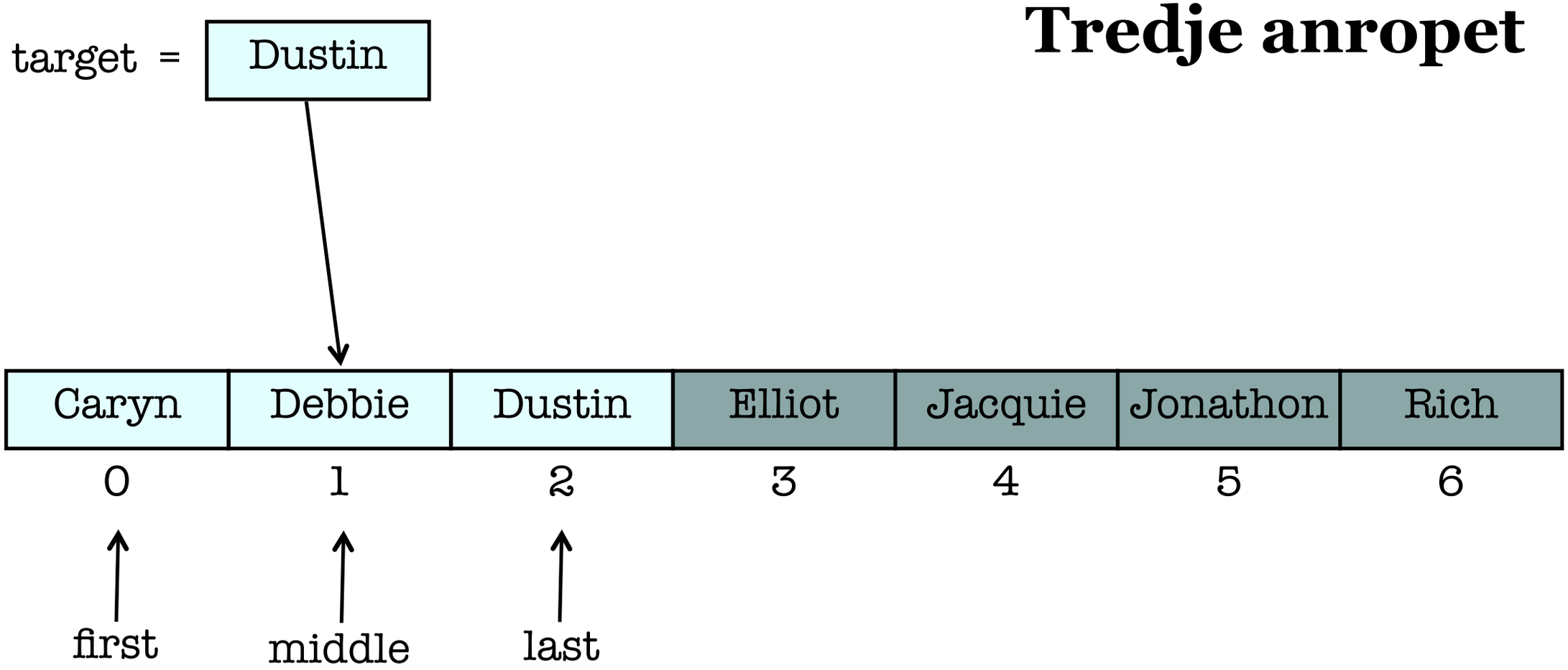
Binärsökning, exempel

Andra anropet



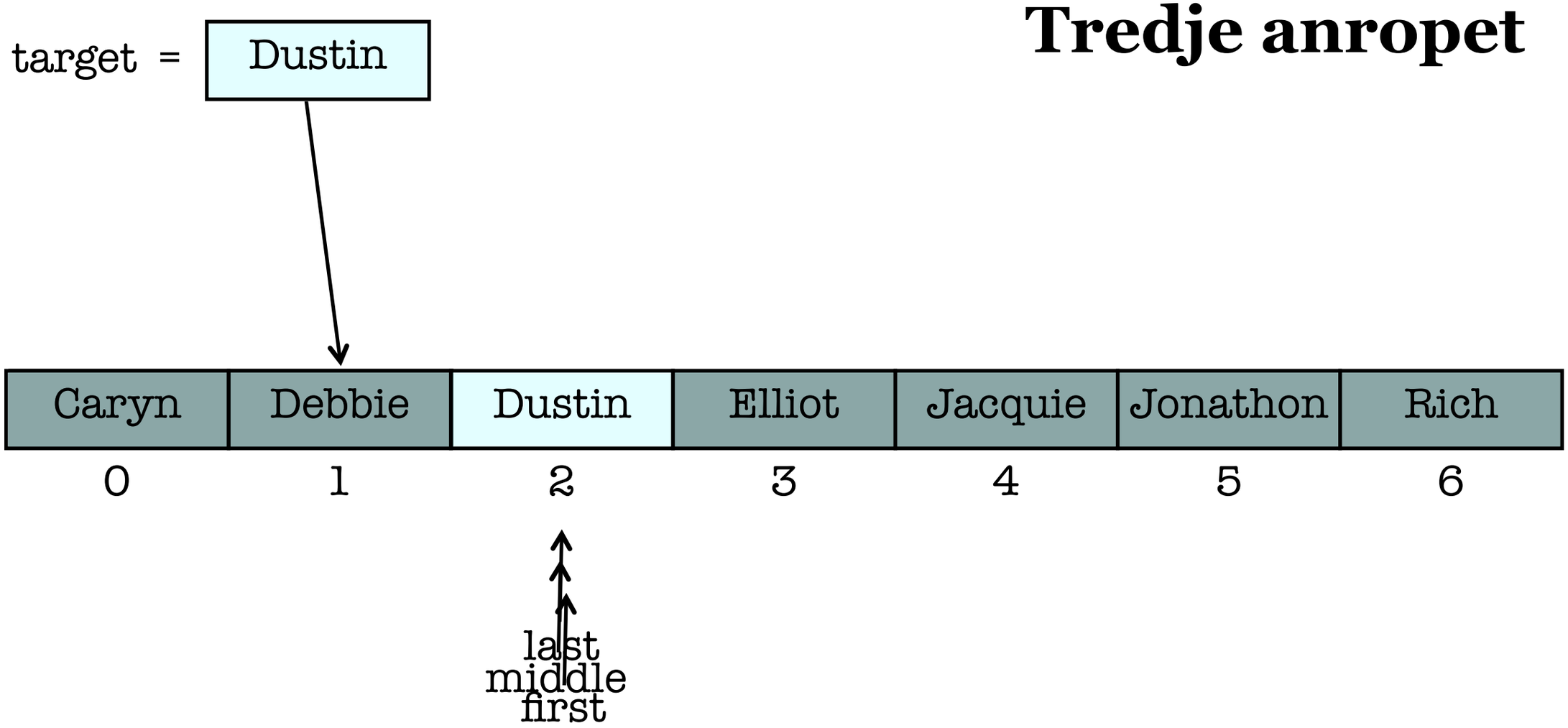
Binärsökning, exempel

Tredje anropet



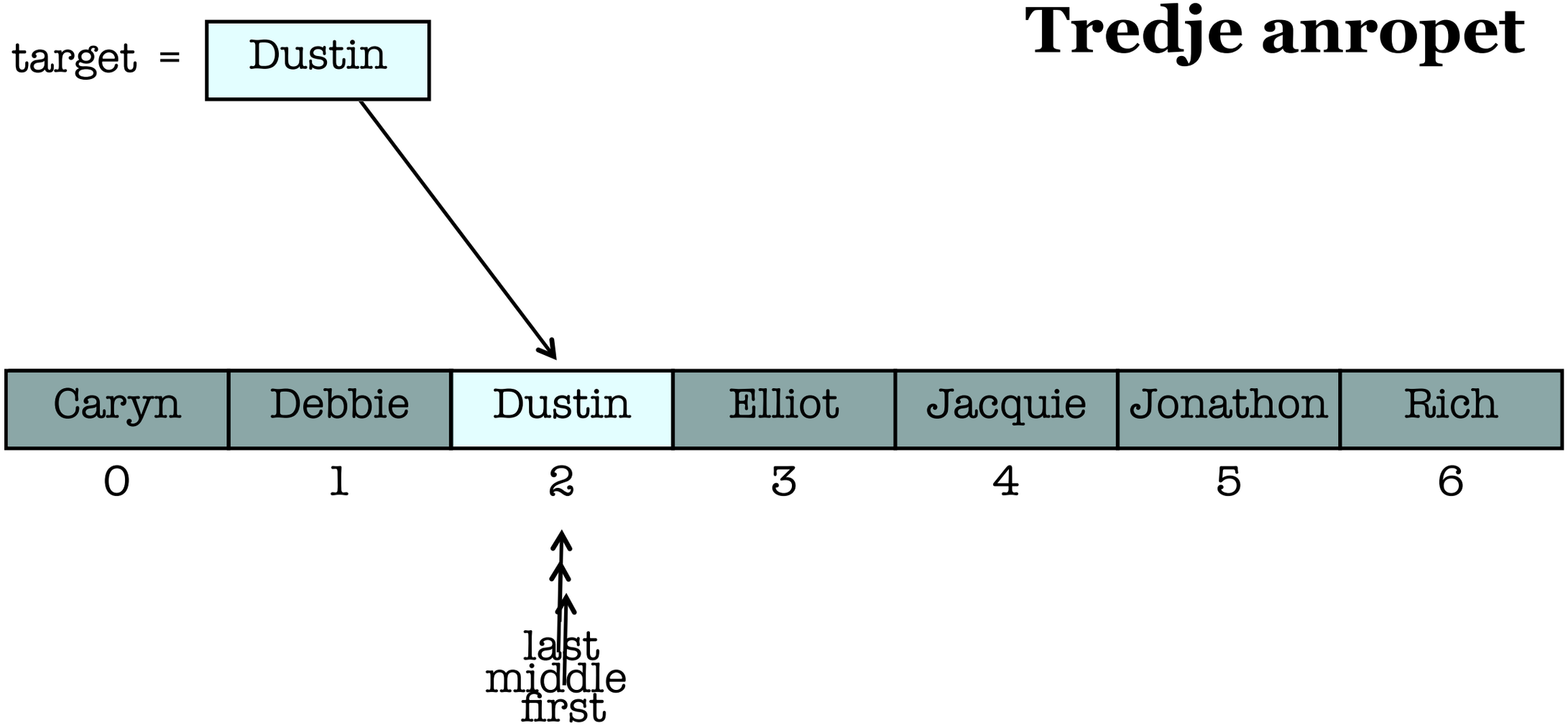
Binärsökning, exempel

Tredje anropet



Binärsökning, exempel

Tredje anropet



Binärsökning, iterativt

Binärsökning är fortfarande enkelt att implementera iterativt.

Iterativ algoritm *binarySearch*(E[] **A**, E *target*):

- *first* = 0, *last* = **A**.*length* − 1
- repetera ända tills *first* > *last*:
  - *middle* = (*first* + *last*) / 2
  - om **A**[*middle*] == *target*: return *middle*
  - om **A**[*middle*] < *target*: *last* = *middle* − 1
  - om **A**[*middle*] > *target*: *first* = *middle* + 1
- return −1

Förvillkor och invariant

Binärsökning har ett viktigt ***förvillkor***:

- fältet måste vara sorterat, annars blir resultatet odefinierat
- sådana förvillkor *måste* skrivas i dokumentationen för en metod/funktion

Under förutsättning att fältet är sorterat så kan vi formulera en ***invariant*** för binärsökningen:

- $first \leq middle \leq last$, och
- $\mathbf{A}[first] \leq target \leq \mathbf{A}[last]$

Att formulera bra invarianter och förvillkor är nödvändigt för att kunna bevisa att algoritmer är korrekta

Binärsökning, effektivitet

Vid varje rekursivt anrop (iterativ loop) eliminerar vi halva fältet

- O(log n)

Ett fält med 16 element kommer alltså att söka igenom delfält med storlekarna 16, 8, 4, 2 och 1 – fem jämförelser i värsta fall:

- $16 = 2^4$
- $5 = \log_2 16 + 1$

Ett dubbelt så stort fält kräver endast 6 jämförelser i värsta fall:

- $32 = 2^5$
- $6 = \log_2 32 + 1$

Ett fält med 32 768 element kräver endast 16 jämförelser!

- $16 = \log_2 32768 + 1$

Arrays.binarySearch

Javas API-klass Arrays innehåller en binärsökning i fält:

- java.util.Arrays.binarySearch(`E[]`, `E`)
- anropas med sorterade fält
- om elementen inte är jämförbara, eller om fältet inte är sorterat, så är resultatet odefinierat
  - dvs, det kastar **inte** ett Exception!
- om det finns flera kopior av samma objekt, så vet vi inte vilken som kommer att hittas

Avancerad rekursion

Hittills har vi tittat på enkla rekursioner

- med precis lika enkla (eller enklare) iterativa alternativ
- de rekursiva anropen motsvarar loopar

Mer komplexa rekursiva algoritmer är inte lika enkla att översätta iterativt

- de behöver en stack (eller någon annan datastruktur) för att lagra de rekursiva anropen
- (eller flera stackar, eller en stack av stackar…)

En stack istället för rekursion

Medelkomplex rekursion kräver en explicit stack för att kunna översättas iterativt:

- push() motsvaras av ett rekursivt anrop
- pop() motsvaras av att returnera
- rekursionen använder sig av Javas egen call stack, medan den iterativa varianten måste använda en egen stack

Vilket sätt som är enklast är en fråga om tycke och smak:

- iteration med en explicit stack
- en implicit stack genom rekursiva anrop

Det finns rekursiva algoritmer där det inte räcker med en enda stack, utan det istället behövs en trädstruktur

Balanserade parenteser, iterativt

```
private static final String PARENS  = "()[]{}";

public static void parseParensIterative(String str) throws ParseException {
        Stack<Integer> parenStack = new Stack<Integer>();
        for (int i = 0; i < str.length(); i++) {
                char c = str.charAt(i);
                int paren = PARENS.indexOf(c);
                if (paren >= 0) {
                        if (paren % 2 == 0) {
                                parenStack.push(paren);
                        } else {
                                if (parenStack.empty())
                                        throw new ParseException("Too many close parentheses", i);
                                int openParen = parenStack.pop();
                                if (paren != openParen + 1)
                                        throw new ParseException("Unbalanced parentheses", i);

                        }
                }
        }
        if (!parenStack.empty())
                throw new ParseException("Too many open parentheses", str.length());

}
```

Balanserade parenteser, rekursivt

```
public static void parseParensRecursive(String str) throws ParseException {
        parseParensRecursive(str, 0, -1);
}

private static int parseParensRecursive(String str, int i, int openParen) throws ParseException {
        Stack<Integer> parenStack = new Stack<Integer>();
        for (int i = 0; i < str.length(); i++) {
                char c = str.charAt(i);
                int paren = PARENS.indexOf(c);
                if (paren >= 0) {
                        if (paren % 2 == 0) {
                                i = parseParensRecursive(str, i+1, paren);
                        } else {
                                if (!parenStack.empty())  if (openParen < 0)
                                        throw new ParseException("Too many close parentheses", i);
                                int openParen = parenStack.pop();
                                if (paren != openParen + 1)
                                        throw new ParseException("Unbalanced parentheses", i);
                                return i;
                        }
                }
        }
        if (!parenStack.empty())  if (openParen >= 0)
                throw new ParseException("Too many open parentheses", i);
        return i;
}
```

Tornen i Hanoi

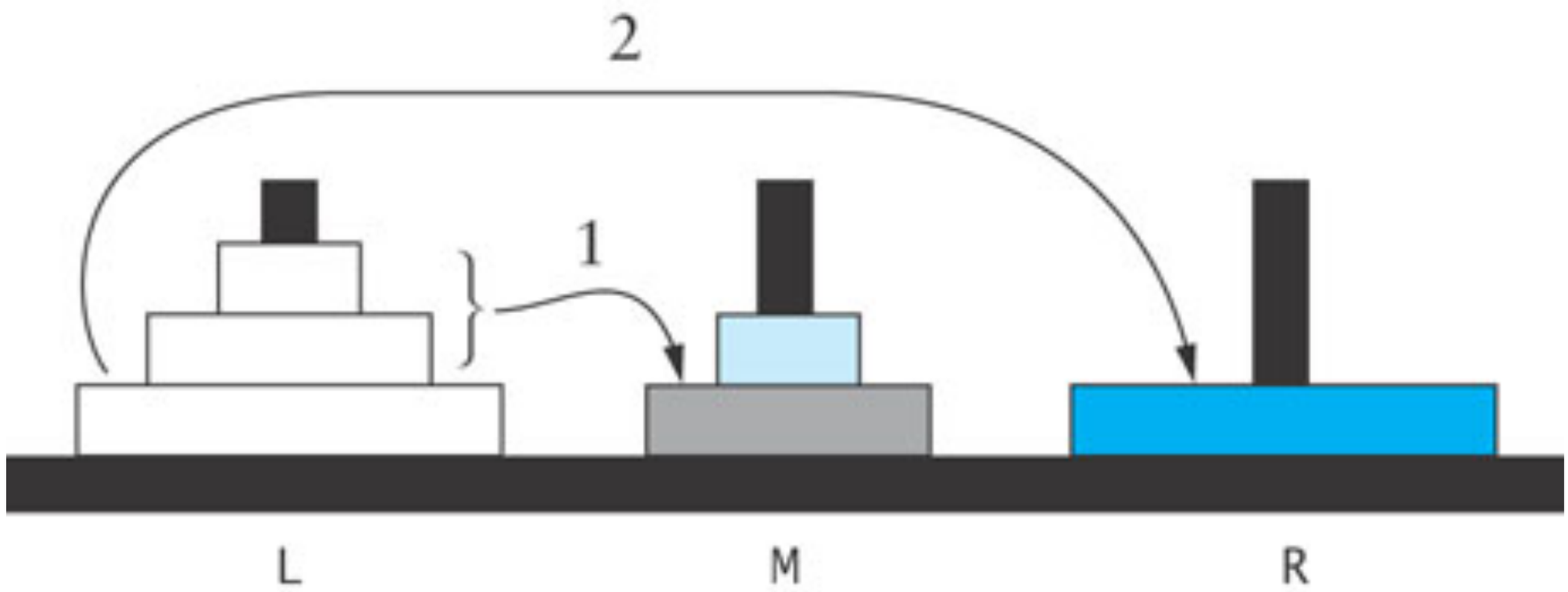
Du ska flytta alla brickor till en annan pinne:

- endast en bricka i taget (den översta) får flyttas
- en större bricka får inte placeras på en mindre

Algoritm för tornen i Hanoi

Lösning till 3-bricksproblemet: att flytta de tre brickorna från pinne L till pinne R.

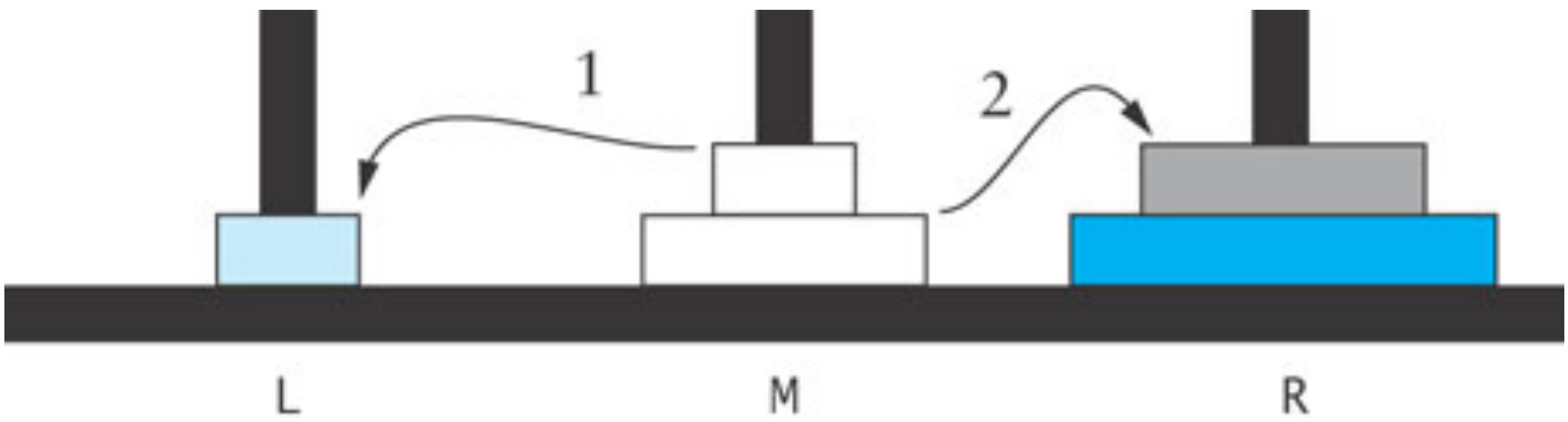
(1) flytta de två översta brickorna från pinne L till pinne M
(2) flytta den understa brickan från pinne L till pinne R
(3) flytta de två översta brickorna från pinne M till pinne R



Algoritm för tornen i Hanoi

Dellösning till 3-bricksproblemet: att flytta de två brickorna från pinne M till pinne R.

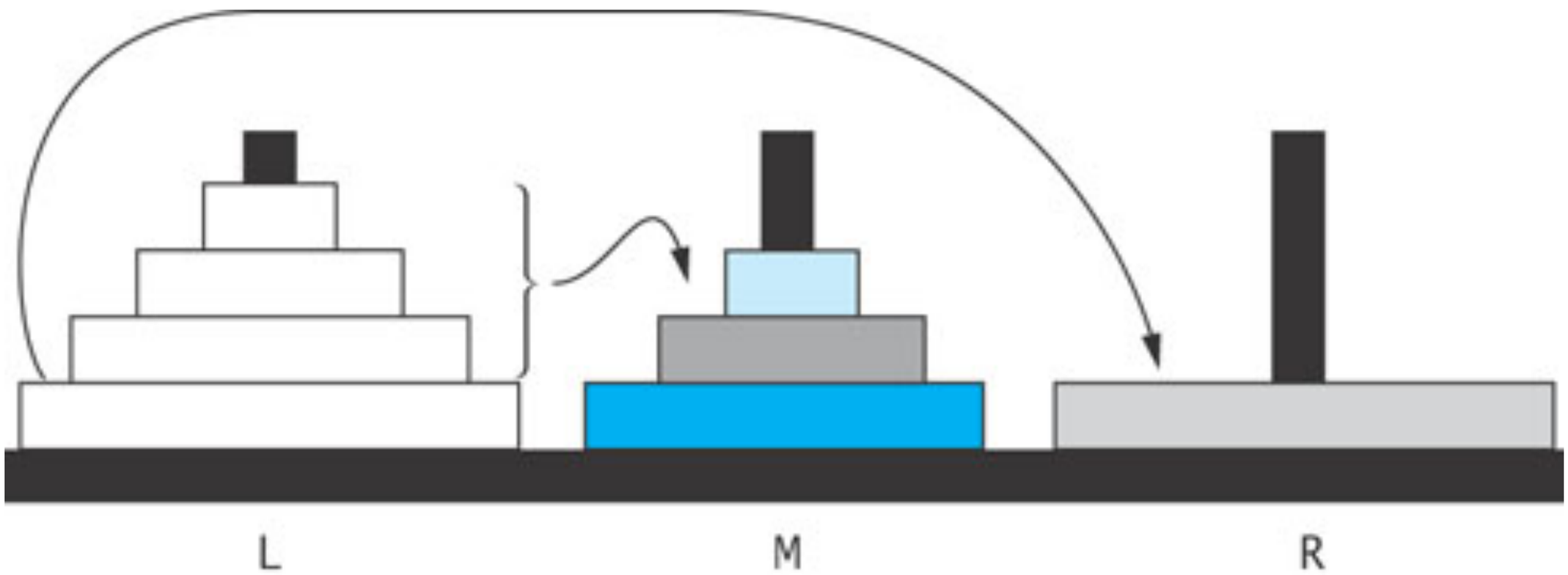
(1) flytta den översta brickan från pinne M till pinne L
(2) flytta den understa brickan från pinne M till pinne R
(3) flytta den översta brickan från pinne L till pinne R



Algoritm för tornen i Hanoi

Lösning till 4-bricksproblemet: att flytta de fyra brickorna från pinne L till pinne R.

(1) flytta de tre översta brickorna från pinne L till pinne M
(2) flytta den understa brickan från pinne L till pinne R
(3) flytta de tre översta brickorna från pinne M till pinne R



Tornen i Hanoi, rekursivt

Rekursiv algoritm för n-bricksproblement, att flytta n brickor från pinne A till pinne B, med mellanpinne C.

Om $n = 1$:

(1) flytta bricka nr 1 (den minsta brickan) från A till B

Annars:

(2) flytta de $n-1$ översta brickorna från A till C
(3) flytta bricka n från A till B
(4) flytta de $n-1$ översta brickorna från C till A

Backtracking

Backtracking är ett sätt att leta efter en lösning genom systematiskt pröva alla möjligheter

T.ex. när du letar efter en väg genom en labyrint:

- när du stöter på en vägkorsning så prövar du först det ena alternativet
- om det misslyckas prövar du nästa alternativ
- fortsätt så för varje vägkorsning i labyrinten

Hitta en stig genom en labyrint

Problembeskrivning:

- labyrinten är en matris med färgade celler
- startpunkten är övre vänstra hörnet (0, 0)
- utgången är nedre högra hörnet (n–1, m–1)
- alla öppna celler har färgen **ÖPPEN**
- alla väggar har färgen **VÄGG**

Lösningsskiss:

- vi prövar rekursivt alla grannar
- celler som vi har besökt får färgen **BESÖKT**
- om cellen är en vägg eller redan besökt så misslyckas vi och backtrackar genom att pröva en tidigare granne
- om vi hittar en stig så ger vi den färgen **STIG**

Rekursiv algoritm för att hitta genom en labyrint

Rekursiv algoritm *hittaStig(cell)*

- om *cell* är utanför labyrinten,
  eller har färgen **VÄGG** eller **BESÖKT**:
  - **return false**
- om *cell* är labyrintens utgång:
  - ge *cell* färgen **STIG**
  - **return true**
- ge *cell* färgen **BESÖKT**
- för varje *granne* till *cell*:
  - om *hittaStig(granne)*:
    ge *cell* färgen **STIG**
    return true
- **return false**

Rekursiv algoritm för att hitta genom en labyrint

Rekursiv algoritm *hittaStig(cell)*

- om *cell* är utanför labyrinten,
  eller har färgen **VÄGG** eller **BESÖKT**:
  - **return false**
- om *cell* är labyrintens utgång:
  - ge *cell* färgen **STIG**
  - **return true**
- ge *cell* färgen **BESÖKT**
- för varje *granne* till *cell*:
  - om *hittaStig(granne)*:
    ge *cell* färgen **STIG**
    return true
- **return false**

rekursivt anrop

Rekursiv algoritm för att hitta genom en labyrint

Rekursiv algoritm *hittaStig(cell)*

- om *cell* är utanför labyrinten,
  eller har färgen **VÄGG** eller **BESÖKT**:
  - **return false**
- om *cell* är labyrintens utgång:
  - ge *cell* färgen **STIG**
  - **return true**
- ge *cell* färgen **BESÖKT**
- för varje *granne* till *cell*:
  - om *hittaStig(granne)*:
    ge *cell* färgen **STIG**
    return true
- **return false**

backtracking

rekursivt anrop

n-queens puzzle

Problembeskrivning:

- att ställa ut *n* schackdamer på ett *n*×*n*-bräde, så att ingen dam hotar någon annan dam
  - http://en.wikipedia.org/wiki/Eight_queens_puzzle

Lösningsförslag:

- för att fylla rad *k*:
  - placera drottningen på någon kolumn
  - försök fylla rad *k+1* (rekursivt)
  - om det misslyckas, placera ut drottningen på en ny kolumn (backtracking)

damproblemet i Java

```
private int[] queens;

public int[] solveQueens(int N) {
        queens = new int[N];
        solveQueensRow(0, N);
        return queens;
}

private boolean solveQueensRow(int n, int N) {
        if (n == N) {
                return true;
        } else {
                for (int i = 0; i < N; i++) {
                        queens[n] = i;
                        if (isConsistent(queens, n))
                                if (solveQueensRow(n+1, N))
                                        return true;
                }
        }
        return false;
}
```

kod lånad från: http://introcs.cs.princeton.edu/java/23recursion/Queens.java.html

damproblemet i Java

```
private int[] queens;

public int[] solveQueens(int N) {
        queens = new int[N];
        solveQueensRow(0, N);
        return queens;
}

private boolean solveQueensRow(int n, int N) {
        if (n == N) {
                return true;
        } else {
                for (int i = 0; i < N; i++) {
                        queens[n] = i;
                        if (isConsistent(queens, n))
                                if (solveQueensRow(n+1, N))
                                        return true;
                }
        }
        return false;
}
```

rekursivt anrop

kod lånad från: http://introcs.cs.princeton.edu/java/23recursion/Queens.java.html

damproblemet i Java

```
private int[] queens;

public int[] solveQueens(int N) {
        queens = new int[N];
        solveQueensRow(0, N);
        return queens;
}

private boolean solveQueensRow(int n, int N) {
        if (n == N) {
                return true;
        } else {
                for (int i = 0; i < N; i++) {
                        queens[n] = i;
                        if (isConsistent(queens, n))
                                if (solveQueensRow(n+1, N))
                                        return true;
                }
        }
        return false;
}
```

backtracking

rekursivt anrop

kod lånad från: http://introcs.cs.princeton.edu/java/23recursion/Queens.java.html

damproblemet i Java (forts.)

```
public boolean isConsistent(int[] queens, int n) {
        for (int i = 0; i < n; i++) {
                if (queens[i]  ==  queens[n])           return false;
                if (queens[i] – queens[n]  ==  n – i)   return false;
                if (queens[i] – queens[n]  ==  i – n)   return false;
        }
        return true;
}

public void printQueens(int[] q) {
        int N = queens.length;
        for (int i = 0; i < N; i++) {
                for (int j = 0; j < N; j++) {
                        if (queens[i] == j)   System.out.print("Q ");
                        else                  System.out.print("· ");
                }
                System.out.println();
        }
        System.out.println();
}
```

damproblemet i Java (forts.)

```
public boolean isConsistent(int[] queens, int n) {
        for (int i = 0; i < n; i++) {
                if (queens[i]  ==  queens[n])           return false;
                if (queens[i] – queens[n]  ==  n – i)   return false;
                if (queens[i] – queens[n]  ==  i – n)   return false;
        }
        return true;
}
```

samma kolumn

```
public void printQueens(int[] q) {
        int N = queens.length;
        for (int i = 0; i < N; i++) {
                for (int j = 0; j < N; j++) {
                        if (queens[i] == j)   System.out.print("Q ");
                        else                  System.out.print(" · ");
                }
                System.out.println();
        }
        System.out.println();
}
```

damproblemet i Java (forts.)

```
public boolean isConsistent(int[] queens, int n) {
        for (int i = 0; i < n; i++) {
                if (queens[i]  ==  queens[n])          return false;
                if (queens[i] – queens[n]  ==  n – i)  return false;
                if (queens[i] – queens[n]  ==  i – n)  return false;
        }
        return true;
}
```

samma kolumn

ena diagonalen

```
public void printQueens(int[] q) {
        int N = queens.length;
        for (int i = 0; i < N; i++) {
                for (int j = 0; j < N; j++) {
                        if (queens[i] == j)  System.out.print("Q ");
                        else                 System.out.print(" · ");
                }
                System.out.println();
        }
        System.out.println();
}
```

damproblemet i Java (forts.)

```
public boolean isConsistent(int[] queens, int n) {
        for (int i = 0; i < n; i++) {
                if (queens[i]  ==  queens[n])          return false;
                if (queens[i] – queens[n]  ==  n – i)  return false;
                if (queens[i] – queens[n]  ==  i – n)  return false;
        }
        return true;
}
```

samma kolumn

ena diagonalen

andra diagonalen

```
public void printQueens(int[] q) {
        int N = queens.length;
        for (int i = 0; i < N; i++) {
                for (int j = 0; j < N; j++) {
                        if (queens[i] == j)  System.out.print("Q ");
                        else                 System.out.print(" · ");
                }
                System.out.println();
        }
        System.out.println();
}
```