4. Recursion, part 2

Programming and Algorithms II

Degree in Bioinformatics

Fall 2018

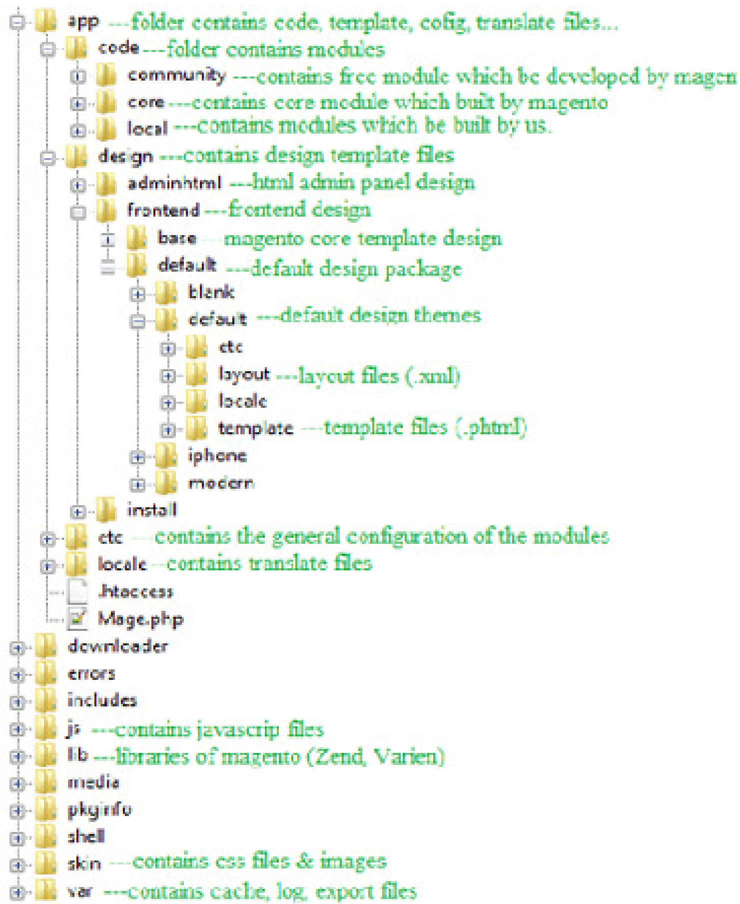
Problem: Traversing a directory

Given a path, list all the files in the folder identified by the path…

… and its subfolders

… and their subfolders

… and their subfolders



Option 1: import os.walk

Option 2, for learning purposes:

recursive program

Problem: Traversing a directory

Tools:

```
from os import listdir
from os.path import isfile, join, isdir
```

listdir(path) returns a list of files+folders in path

join(path,filename) returns path/filename

(or path\filename in Windows)

isfile(string) tells whether string is a file (with path)

(if not, let's say it's a directory)

Traversing a directory

```
from os import listdir
from os.path import isfile, join, isdir

def printAllFiles(root):
    for f in listdir(root):
        ff = join(root,f)
        if isfile(ff):
            print(ff)
        else: # it is a directory
            printAllFiles(ff)
```

Traversing a directory, better

```
from os import listdir
from os.path import isfile, join, isdir

def printAllFiles(root,n,ind):
     for f in listdir(root):
          ff = join(root,f)
          if isfile(ff):
               print(" "*ind*n,ff)
          else: # it is a directory
               print(" "*ind*n,"FOLDER ",ff)
               printAllFiles(ff,n+1,ind)
```

Merging two sorted lists

Given two lists that are sorted, compute a list with their unión

[1, 2, 2, 5, 6, 6, 9, 10, 10, 12]

[0, 2, 4, 5, 5, 7, 8, 9, 9, 11, 12]

→

[0,1,2,2,2,4,5,5,5,6,6,7,8,9,9,9,10,10,11,12,12]

Merging two sorted lists

```
def merge(lst1, lst2):
    i = 0
    j = 0
    result = []
    while i < len(lst1) and j < len(lst2):
        if lst1[i] <= lst2[j]:
            result.append(lst1[i])
            i = i+1
        else:
            result.append(lst2[j])
            j = j+1
(… continues …)
```

```
(… continued …)

    result.extend(lst1[i:])
    result.extend(lst2[j:])

    return result
```

At every iteration, we move either one element from lst1 or from lst2

Loop body is O(1)

Time O(len(lst1) + len(lst2))

Mergesort

Idea:

Given that merging two sorted lists is easy…

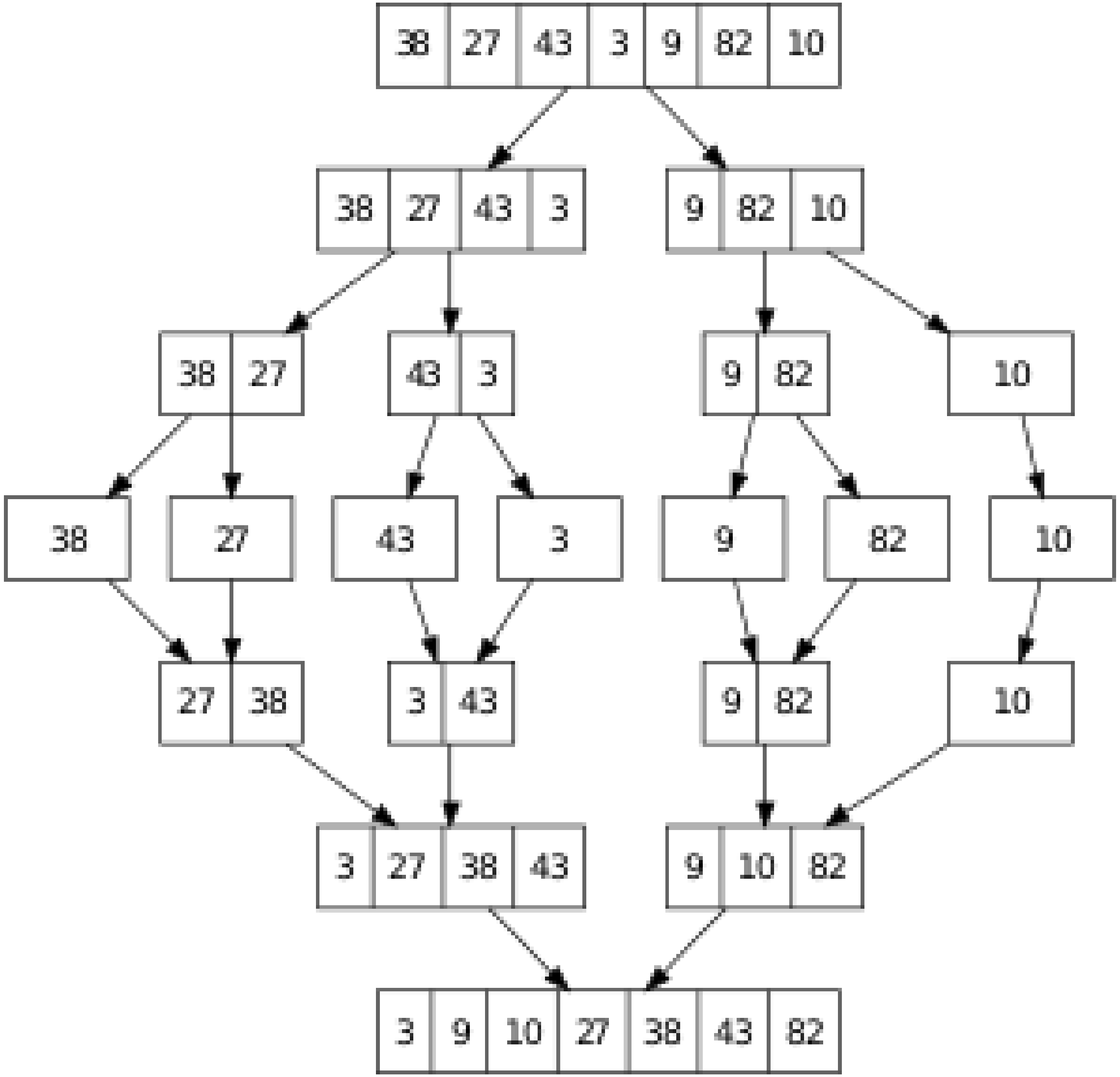
1. Split your big list into two lists
2. Sort each one separately
3. Merge the resulting sorted lists

Base case: list is sufficiently small to sort some other way

Mergesort

```
def mergeSort(lst):
    if len(lst) <= 1:
        return lst
    else:
        mid = (len(lst)+1)//2    # +1 is optional
        lefthalf = mergeSort(lst[:mid])
        righthalf = mergeSort(lst[mid:])
        return merge(lefthalf,righthalf)
```

Mergesort


38 27 43 3 9 82 10
38 27 43 3
9 82 10
38 27
43 3
9 82
10
38
27
43
3
9
82
10
27 38
3 43
9 82
10
3 27 38 43
9 10 82
3 9 10 27 38 43 82

Mergesort

Each “row” deals with varying number of arrays

But time to deal with an array O(its length)

Total sum of lengths of arrays is length of original array

-> total work “per row” O(n)

Number of rows: log n in top, log n in bottom

-> Total time O(n log n)

Uses additional memory, not in-place sorting

With some care, the copying of extra lists can be optimized

Good way of sorting sequential files in external memory

Multiplying large numbers

Given: two “large” integers x, y

Return: their product x*y

(in many programming languages, int’s are limited to some fixed range e.g. [-2^{32}…2^{32}]; overflow is produced if exceded; Python automatically extends int’s to be as large as required, BigInts)

Multiplying large numbers

Think of integers as list of bits for a moment

Assumption:

- Sum is linear in #bits of operands
- Multiplying by 2^i is fast – add i 0's at the end
- Dividing by 2^i is fast – drop last i bits
- O(#digits + i) time

In computers, hardware directly supports product and division by powers of 2 (shifts)

Multiplying large numbers

School algorithm (x,y)

```
sum = 0
for i  in [0 … numdigits(y)-1]
    sum += x * y[i] * 2^i
```

Product by a digit y[i] and by 2^i is O(numdigits(x))

$O(n^2)$ if both x and y have n digits (and B constant)

Recursively

Numbers with 2k digits

$(x1*2^k + x0) * (y1*2^k + y0) =$

$x1*y1 * 2^{2k} + (x1*y0 + x0*y1) * 2^k + x0*y0$

Two n-bit numbers →

4 products of n/2 bit numbers

+ 3 sums + 2 shifts

Claim: same * of digits as before before -> $O(n^2)$

The Karatsuba – Ofman trick

Karatsuba Ofman method

We did $(x1*2^k + x0) * (y1*2^k + y0) =$

$x1*y1 * 2^{2k} + (x1*y0 + x0*y1) * 2^k + x0*y0$

$a = x1*y1$

$b = x0*y0$

$c = (x1+x0)*(y1+y0)$ $(= x1*y1+x1*y0+x0*y1+x0*y0)$

$c = c – a – b$ $(= x1*y0 + x0*y1)$

$result = a*2^{2k} + c*2^k + b$

3 multiplications of n/2 bit numbers, 6 sums, 2 shifts

The Karatsuba Ofman method

Let $T(n)$ be running time on n bit numbers

We must have $T(n) = O(n) + 3\,T(n/2)$ "Recurrence"

Assume $T(n) \approx n^{\alpha}$ $(1 <= \alpha <= 2)$

We must have $n^{\alpha} \approx O(n) + 3\,(n/2)^{\alpha}$

So $3 / 2^{\alpha} \approx 1$, so $\alpha = \log_2(3) \approx 1.585...$

Running time is $O(n^{1.585})$

Even better method

Schonhäge-Strassen's algorithm. Very cool

Based on Discrete Fourier Transform

Starts like mergesort, then gets complex

O(n log (n) log log(n)) time

Beats Karatsuba Ofman for 100,000's of bits or so

Can we do O(n) (like sum)? We don't know

Printing all subsets

```
Given n, print one line with every subset of
{1..n}, in list format. In any order.
For example, for n=3

[]
[1]
[2]
[3]
[1,2]
[1,3]
[2,3]
[1,2,3]
```

Printing all subsets

Many solutions

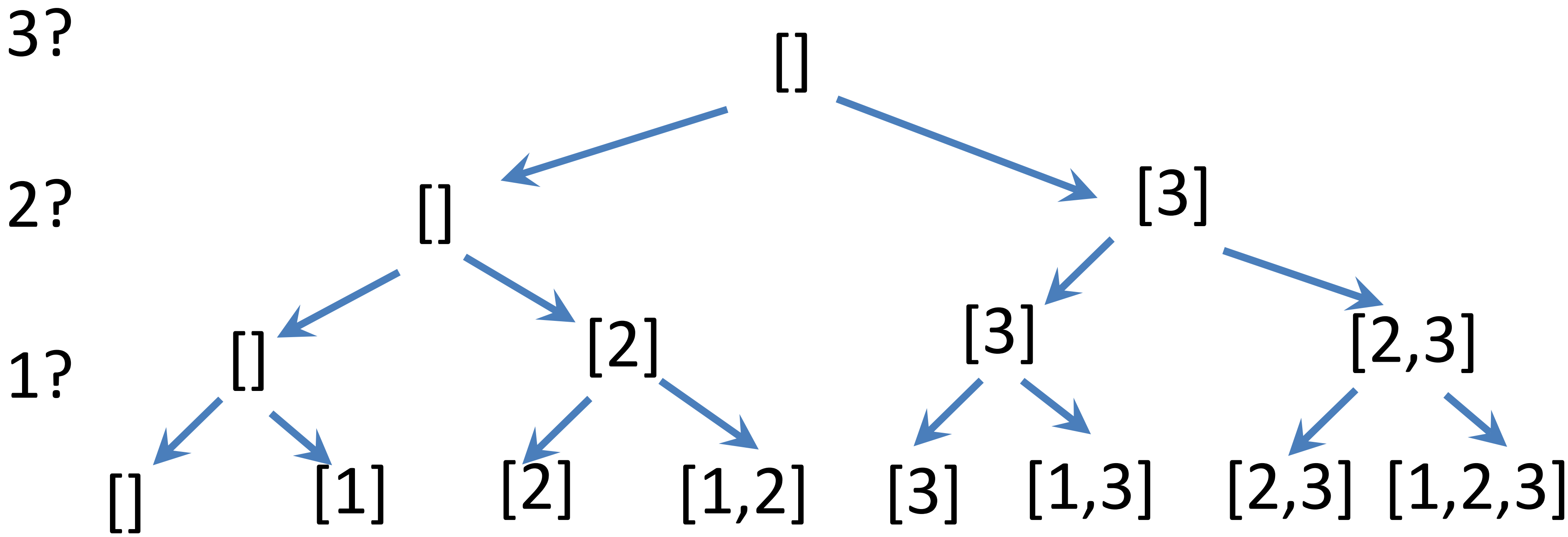
Some iterative, some recursive

A recursive one is obtained by considering that if S is a subset and x in S, there are two kinds of subsets of S:

- Those that do not contain x
- Those that contain x

Example: without 3: [], [1], [2], [1,2]
with 3: [3], [1,3], [2,3], [1,2,3]

Printing all subsets



Printing all subsets

```
def subsets(lstin,lstout):
    "prints all sets of the form (a subset of lstin)+lstout"
    "lstout is the set of elements we have already decided"
    "to include in the subset to print in this execution"
    if len(lstin) == 0:
        print(lstout)
    else:
        x = lstin[len(lstin)-1]
        lstin1 = lstin[0:len(lstin)-1]
        lstout1 = lstout.copy()
        subsets(lstin1,lstout1)
        lstout1 = [x] + lstout1
        subsets(lstin1,lstout1)
```

```
[]
[1]
[2]
[1,2]
[3]
[1,3]
[2,3]
[1,2,3]
```

Inefficiencies because of copy, :, +
Think how to reduce using indices

The knapsack problem

The knapsack problem

You find a treasure chest

full of precious objects

Each object i has a value v_i and a weight w_i

Your backpack can hold up to weight W, or it breaks

Which is the most valuable subset of objects

that you can take home?

The knapsack problem

Example:

Max capacity is W=16

	A	B	C	D	E	F	G
Value	7	9	5	12	14	6	12
Weight	3	4	2	6	7	3	5

(One) best combination is to take objects A, C, D, G

which has value 7+5+12+12 = 36

and weight 3+2+6+5 = 16 (full backpack)

The knapsack problem

Explore all subsets as before

If a subset:

– has weight <= W

– has value better than any best seen before

then keep it

Optimization: don't explore extensions of subsets that already have weight >W

The knapsack problem

Another way of looking at it:

Suppose we know how to solve the problem with n objects
(find best combination of the objects with some max weight W)

Now we have n objects, plus 1 more
The best combination is the best of the following two:

- the best combination of the n first objects

or

- picking the n+1-th object, plus the best combination of the n objects with weight <= W-weight(n+1)

The knapsack problem

def knapsack(v,w,i,W)

returns a tuple (best,bestv,bestw) such that

- best is a list containing a subset of [0..i]
- it is the highest-value combination of items in [0..i] that has weight at most W
- the value of best is bestv
- the weight of best is bestw (<= W)

Initial call: knapsack(v,w,len(v)-1,W)

The knapsack problem

```
def knapsack(v,w,i,W):
     "definition as before"
     if i == -1:
        return ([],0,0)
    else:
        # best combination without including item i
        best, bestv, bestw = knapsack(v,w,i-1,W)
        if w[i] <= W:
            # if item i fits in knapsack,
            # aim for best combination that leaves space for item i
            best1, bestv1, bestw1 = knapsack(v,w,i-1,W-w[i])
            if bestv1 + v[i] > bestv:
                best1.append(i)
                best, bestv, bestw =
                         best1, bestv1 + v[i], bestw1 + w[i]
        return (best, bestv, bestw)
```

Can we do better?

Time is O(number of subsets of {1..n}) = $O(2^n)$. Bad!

Can't do much better in general.

- Knapsack is one of (many) NP-complete problems
- All known solutions for all NP-complete problems are exponential
- We believe no subexponential solution exists for any of them
- If you have a subexponential solution for one NP-complete problem, you have subexponential solutions for all NP-complete problems

Trading time for memory

Note that the algorithm we gave solves the same subproblems again and again, with different W

- Grow solutions with weight 1, 2, 3, …, up to W
- Use a dictionary to remember problems already solved?
- Time O(nW). Good if W is small, still bad if W is large

This is called Dynamic Programming: remembering work you did to avoid recomputing

More next quarter

Greedy approximation algorithm

A reasonable heuristic:

- Pick up the object with highest ratio v/w   (euros/kg)
- Repeat

O(n log n) time

A small modification of this guarantees a solution with value at least 0.5* (optimum solution)

This is called greedy algorithm: do what seems best right now (locally), hoping that it leads to a good global solution

A more complicated solution changes 0.5 to any a < 1

Greedy approximation algorithm (2)

```
def greedyKnapsack(v,w,W):

    ratio = [(i,v[i]/ w[i]) for i in range(len(v))]
    ratio.sort(key=lambda iv: iv[1],reverse=True)

    bestv = 0
    bestw = 0
    best = []

    for i in range(len(ratio)):
        item, rat = ratio[i]
        if bestw + w[item] <= W:
            best.append(item)
            bestv += v[item]
            bestw += w[item]

    best.sort()
    return (best,bestv,bestw)
```

Examples

```
# problem in slides
v = [7,9,5,12,14,6,12]
w = [3,4,2,6,7,3,5]
W = 16
print(knapsack(v,w,len(v)-1,W))
>>> ([0, 2, 3, 6], 36, 16)
print(greedyKnapsack(v,w,W))
>>> ([0, 1, 2, 6], 33, 14)

# instance P07 from
# https://people.sc.fsu.edu/~jburkardt/datasets/knapsack_01/knapsack_01.html
v = [135, 139, 149, 150, 156, 163, 173, 184, 192, 201, 210, 214, 221, 229, 240]
w = [70, 73, 77, 80, 82, 87, 90, 94, 98, 106, 110, 113, 115, 118, 120]
W = 750
print(knapsack(v,w,len(v)-1,W))
>>>  ([0, 2, 4, 6, 7, 8, 13, 14], 1458, 749)
print(greedyKnapsack(v,w,W))
>>>  ([0, 1, 2, 6, 7, 8, 13, 14], 1441, 740)
```

Sudoku

```
solve_sudoku(current_box)
    if all the boxes are filled:
        return true # solved!
    else:
        if current_box is filled:
            return solve_sudoku(next box)
        else:   # try all 9 possible numbers
            for number in 1 to 9
                if that number is 'valid'
                # (meaning: okay to put in box)
                    try that number in current_box
                    if we can "solve_sudoku"
                       for the rest of the puzzle:
                        return true
            # if we got here, no number led to a solution:
            return false
```

Sudoku

Sudoku players know a lot of tricks to discard many options

But guess what:

A generalization of sudoku to nxn is NP-complete

= no mater how many tricks you add, a Sudoku solver will explore exponentially many configurations in the worst case

(unless all NP-complete problems are easier than we think)

Perhaps not 9^{nxn}… perhaps “just” 1.3^{nxn}. Still bad

Conclusions

- Multiple recursion is a powerful tool
- Elegant solutions to complex problems
  - For example, exhaustive search problems
- Some problems seem to be inherently exponential
  - many many bioinformatics problems are NP-complete
  - (they'll tell you they're "NP-hard": slight technical difference)
  - think of heuristics…. and good luck
  - monsters worse than NP-hard exist out there