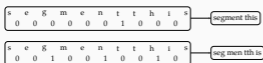


Enumerating segmentations

sketch of a non-recursive solution



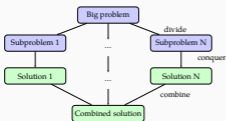
- '1' means there is a boundary at this position
- Problem is now enumerating all possible binary strings of length $n - 1$ (this is binary counting)

Divide and conquer

- The general idea is dividing the problem into smaller parts until it becomes trivial to solve
- Once small parts are solved, the results are combined
- Goes well with recursion
- We have already seen a particular flavor: binary search
- The algorithms like binary search are sometimes called *decrease and conquer*

Divide and conquer

General idea



Divide and conquer

an example: nearest neighbors (only a sketch)

- Task: find the closest two points
- Direct solution: $20 \times 20 = 400$ comparisons¹
- Divide
- Solve separately (conquer): $10 \times 10 + 10 \times 10 = 200$ comparisons
- Combine: pick the minimum of the individual solutions



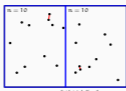
- Gain is higher when n is larger, and we divide further

¹Presumably, $(20 \times 19) / 2 = 190$. In this class we focus on 'order' of operations, rather than the exact numbers. And, the order of gains by divisions is the same.

Divide and conquer

an example: nearest neighbors (only a sketch)

- Task: find the closest two points
- Direct solution: $20 \times 20 = 400$ comparisons¹
- Divide
- Solve separately (conquer): $10 \times 10 + 10 \times 10 = 200$ comparisons
- Combine: pick the minimum of the individual solutions



- Gain is higher when n is larger, and we divide further

¹Presumably, $(20 \times 19) / 2 = 190$. In this class we focus on 'order' of operations, rather than the exact numbers. And, the order of gains by divisions is the same.

Divide and conquer

summary

- This is probably the most common pattern
- Divide and conquer does not always yield good results, the cost of merging should be less than the gain from the division(s)
- Many of the important algorithms fall into this category:
 - merge sort and quick sort (coming soon)
 - integer multiplication
 - matrix multiplication
 - fast Fourier transform (FFT)

Greedy algorithms

- An algorithm is greedy if it optimizes a local constraint
- For some problems, greedy algorithms result in correct solutions
- In others they may result in 'good enough' solutions
- If they work, they are efficient
- An important class of graph algorithms fall into this category (e.g., finding shortest paths, scheduling)

Greedy algorithms

a simple example: 'change making'

- We want to produce minimum number of coins for a particular sum s
 - Pick the largest coin $c \leq s$
 - set $s = s - c$
 - repeat 1 & 2 until $s = 0$
- Is this algorithm correct?
- Think about coins of 10, 30, 40 and apply the algorithm for the sum value of 60
- Is it correct if the coin values were limited Euro coins?

Dynamic programming

- Dynamic programming is a method to save earlier results to reduce computation
- It is sometimes called memoization (it is not a typo)
- Again, a large number of algorithms we use fall into this category, including common parsing algorithms

Dynamic programming

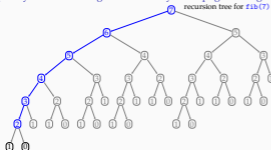
example: Fibonacci

```
def memoFib(n, memo = {0: 0, 1: 1}):
1  if n not in memo:
2      memo[n] = memoFib(n-1) + memoFib(n-2)
3  return memo[n]
```

- We save the results calculated in a dictionary,
- if the result is already in the dictionary, we return without recursion
- Otherwise we calculate recursively as before
- The difference is big, but there is also a 'neater' solution without (explicit) memoization

Complexity of Fibonacci algorithm with dynamic programming

recursion tree for fib(7)



Summary

- We saw a few general approaches to (efficient) algorithm design
- Designing algorithms is not a mechanical procedure: it requires creativity
- There are other common patterns, including
 - Backtracking, branch-and-bound
 - Randomized algorithms
 - Distributed algorithms (sometimes called swarm optimization)
 - Transformation
- Designing algorithms is difficult (possibly, not as difficult as analyzing them)

Next:

- Sorting
- Reading: [goodrich2013](#)

Nearest neighbors

an exercise

- Define and implement a divide-and-conquer algorithm for nearest neighbor problem, which divides the input into two until the solution becomes trivial
- Analyze your algorithm and compare to the naive version sketched above (an implementation was provided in the previous lecture)

Linear search

a little bit of optimization

```
1 def rl_search(seq, val, i=0):
2     if not seq:
3         return None
4     if val == seq[0]:
5         return i
6     else:
7         return rl_search(seq[1:], val,
8                          i+1)
```

```
1 def rl_search2(seq, val, i=0):
2     if i == len(seq):
3         return None
4     if val == seq[i]:
5         return i
6     else:
7         return rl_search2(seq, val, i
8                          + 1)
```

Which one is faster, and why?

Better solutions for Fibonacci numbers

```
1 def fibo(n):
2     if n <= 1:
3         return (n, 0)
4     a, b = fibo(n - 1)
5     return (a+b, a)
```

```
1 def fibo(n):
2     if n <= 1:
3         return n
4     a, b = 0, 1
5     for i in range(0, n):
6         a, b = b, a + b
7     return a
```

Which one is faster/better?

Segmentation

with yield

```
1 def segment_r(seq):
2     if len(seq) == 1:
3         yield [seq]
4     else:
5         for seg in segment_r(seq[1:]):
6             yield [seq[0]] + seg
7             yield [seq[0]] + seg[1:]
```

Acknowledgments, credits, references

- Some of the slides are based on the previous year's course by Corina Dima.