CSCM12: software concepts and efficiency
Some algorithmic design paradigms, sorting algorithms

Cécilia Pradic

Swansea University, 06/20/2023

Today

I will touch on many topics in this lecture

Goals

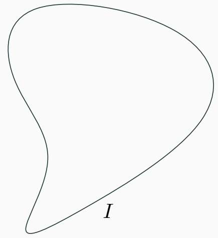
- Introduce divide-and-conquer algorithms
- Mention two other techniques that may be useful: dynamic programming (recalled from last week) and greedy algorithms
- Finally, introduce classical sorting algorithms over arrays

- I will refer back & and expand on this material later

Divide-and-conquer

High-level concept

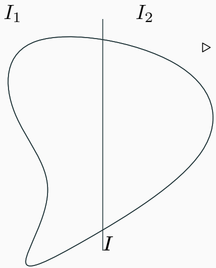
A kind of recursive algorithm where the size of the input is shrunk by a factor in the recursive calls



Divide-and-conquer

High-level concept

A kind of recursive algorithm where the size of the input is shrunk by a factor in the recursive calls

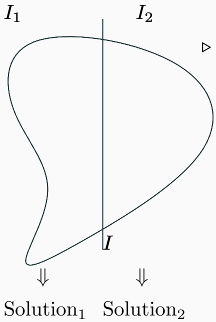


▷ Split the problem into k subparts

Divide-and-conquer

High-level concept

A kind of recursive algorithm where the size of the input is shrunk by a factor in the recursive calls



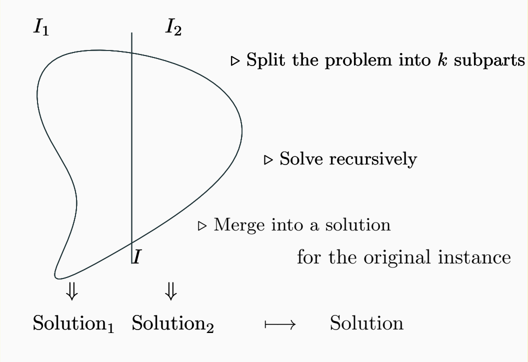
▷ Split the problem into k subparts

▷ Solve recursively

Divide-and-conquer

High-level concept

A kind of recursive algorithm where the size of the input is shrunk by a factor in the recursive calls



Example: dichotomy search

Scenario: imagine you are looking up a word in the dictionary

Example: dichotomy search

Scenario: imagine you are looking up a word in the dictionary

- Do you look-up each word sequentially?

Example: dichotomy search

Scenario: imagine you are looking up a word in the dictionary

- Do you look-up each word sequentially?
- No: start in the middle, and then...

Example: dichotomy search

Scenario: imagine you are looking up a word in the dictionary

- Do you look-up each word sequentially?
- No: start in the middle, and then...

We have already seen this!

Example: dichotomy search

```
/* Assumptions: arr contains an increasing
                sequence of values
                arr[mi] <= 0 and arr[ma] >=0*/
static int dicho_rec(int[] arr, int mi, int ma)
{
   if (ma <= mi)
      return mi;
   final int mid = (ma+mi)/2;
   if (arr[mid] <= 0)
     return dicho_rec(arr,mid,ma);
   else
     return dicho_rec(arr,mi,mid);
}
```

- A good size metric: `ma-mi`
- Size divided by two at each call!

Another example: exponentiation

```java
static double naivePow(double a, int n)
{
  if(n == 0)
    return 1;
  else if(n < 0)
    return 1/naivePow(a,-n);
  else
    return a * naivePow(a, n - 1);
}
```

Complexity: $\mathcal{O}(n)$

Can we do better?

Another problem

Problem

Input: An array A of size n

Output: An element x of A occuring more than $\frac{n}{2}$ times

A naive solution? A divide-and-conquer solution?

Another problem

Problem

Input: An array A of size n

Output: An element x of A occuring more than $\frac{n}{2}$ times

A naive solution? A divide-and-conquer solution?

Naive solution

- Count the number of occurence of an element $\rightarrow \mathcal{O}(n)$

Another problem

Problem

Input: An array A of size n

Output: An element x of A occuring more than $\frac{n}{2}$ times

A naive solution? A divide-and-conquer solution?

Naive solution

- Count the number of occurence of an element $\rightarrow \mathcal{O}(n)$
- Do it for every element of the array

Another problem

Problem

Input: An array A of size n

Output: An element x of A occuring more than $\frac{n}{2}$ times

A naive solution? A divide-and-conquer solution?

Naive solution

- Count the number of occurence of an element $\rightarrow \mathcal{O}(n)$
- Do it for every element of the array $\rightarrow \mathcal{O}(n^2)$

Complexity of divide-and-conquer algorithms

Generic advantages of divide-and-conquer:

- Relatively easy to come up with
- Typically good time complexity
- Easy to parallelize

Complexity of divide-and-conquer algorithms

Generic advantages of divide-and-conquer:

- Relatively easy to come up with
- Typically good time complexity
- Easy to parallelize

$\rightsquigarrow$ How to compute their time complexity?

Complexity of divide-and-conquer algorithms

Generic advantages of divide-and-conquer:

- Relatively easy to come up with
- Typically good time complexity
- Easy to parallelize

$\rightsquigarrow$ How to compute their time complexity?

The typical equation

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

for some $a, b > 0$ and $f : \mathbb{N} \to \mathbb{N}$

Complexity of divide-and-conquer algorithms

Generic advantages of divide-and-conquer:

- Relatively easy to come up with
- Typically good time complexity
- Easy to parallelize

$\rightsquigarrow$ How to compute their time complexity?

The typical equation

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

for some $a, b > 0$ and $f : \mathbb{N} \to \mathbb{N}$

Previous examples:

- $a = 1, b = 2, f = \mathcal{O}(1)$
- $a = 2, b = 2, f = \mathcal{O}(n)$

Quick technicalities

(feel free to ignore on first reading)

- Complexity functions are function $\mathbb{N} \to \mathbb{N}$
- Not a huge deal:
  - As long as the domain is a superset of $\mathbb{N}$ (or an suffix thereof)
  - as long as the function is assumed to dominate/be dominated by the real complexity function
  - another possible hack/reduction

The more precise typical equation

$$T(n) = a'T\left(\left\lceil \frac{n}{b} \right\rceil\right) + a''T\left(\left\lfloor \frac{n}{b} \right\rfloor\right) + f(n)$$

with $a = a' + a''$ typically yield the same asymptotic result up to Θ

Quick technicalities

(feel free to ignore on first reading)

- Complexity functions are function $\mathbb{N} \to \mathbb{N}$
- Not a huge deal:
  - As long as the domain is a superset of $\mathbb{N}$ (or an suffix thereof)
  - as long as the function is assumed to dominate/be dominated by the real complexity function
  - another possible hack/reduction

The more precise typical equation

$$T(n) = a'T\left(\left\lceil \frac{n}{b} \right\rceil\right) + a''T\left(\left\lfloor \frac{n}{b} \right\rfloor\right) + f(n)$$

with $a = a' + a''$ typically yield the same asymptotic result up to Θ

$\to$ it's okay if you are a bit sloppy with rounding at first blush (or only consider inputs whose sizes are powers of b)

A tool to solve many of these recurrences

- Useful to solve many of these (but not all)
- A bit of a bore to remember...

Master theorem

Assume that $T(n) = aT\left(\frac{n}{b}\right) + f(n)$

1. If $f(n) = \mathcal{O}(n^{\log_b(a)-\varepsilon})$ for some $\varepsilon > 0$,
   ▷ then $T(n) = \Theta(n^{\log_b(a)})$
2. If $f(n) = \Theta\left(n^{\log_b(a)} \log(n)^k\right)$ for some $k \geq 0$,
   ▷ then $T(n) = \Theta\left(n^{\log_b(a)} \log(n)^{k+1}\right)$
3. If $f(n) = \Omega\left(n^{\log_b(a)+\varepsilon}\right)$ for some $\varepsilon > 0$,
   and there is $c < 1$ such that $af\left(\frac{n}{b}\right) \leq cf(n)$,
   ▷ then $T(n) = \Theta(f(n))$

Intuitions for the master theorem

Master theorem $(T(n) = aT\left(\frac{n}{b}\right) + f(n))$

1. If $f(n) = \mathcal{O}(n^{\log_b(a)-\varepsilon})$ for some $\varepsilon > 0$,
   ▷ then $T(n) = \Theta(n^{\log_b(a)})$
2. If $f(n) = \Theta\left(n^{\log_b(a)}\log(n)^k\right)$ for some $k \geq 0$,
   ▷ then $T(n) = \Theta\left(n^{\log_b(a)}\log(n)^{k+1}\right)$
3. If $f(n) = \Omega\left(n^{\log_b(a)+\varepsilon}\right)$ for some $\varepsilon > 0$, $\exists c < 1.\ af\left(\frac{n}{b}\right) \leq cf(n)$,
   ▷ then $T(n) = \Theta(f(n))$

Rough idea: does the pre/post-processing time $f(n)$ drive the complexity or the way the recursive calls handled?

Intuitions for the master theorem

Master theorem $\left(T(n) = aT\left(\frac{n}{b}\right) + f(n)\right)$

1. If $f(n) = \mathcal{O}(n^{\log_b(a)-\varepsilon})$ for some $\varepsilon > 0$,
   - ▷ then $T(n) = \Theta(n^{\log_b(a)})$
2. If $f(n) = \Theta\left(n^{\log_b(a)} \log(n)^k\right)$ for some $k \geq 0$,
   - ▷ then $T(n) = \Theta\left(n^{\log_b(a)} \log(n)^{k+1}\right)$
3. If $f(n) = \Omega\left(n^{\log_b(a)+\varepsilon}\right)$ for some $\varepsilon > 0$, $\exists c < 1.\ af\left(\frac{n}{b}\right) \leq cf(n)$,
   - ▷ then $T(n) = \Theta(f(n))$

Rough idea: does the pre/post-processing time $f(n)$ drive the complexity or the way the recursive calls handled?

Our examples

- Dichotomy/fast exponentiation:

Intuitions for the master theorem

Master theorem $(T(n) = aT\left(\frac{n}{b}\right) + f(n))$

1. If $f(n) = \mathcal{O}(n^{\log_b(a)-\varepsilon})$ for some $\varepsilon > 0$,
   - ▷ then $T(n) = \Theta(n^{\log_b(a)})$
2. If $f(n) = \Theta\left(n^{\log_b(a)} \log(n)^k\right)$ for some $k \geq 0$,
   - ▷ then $T(n) = \Theta\left(n^{\log_b(a)} \log(n)^{k+1}\right)$
3. If $f(n) = \Omega\left(n^{\log_b(a)+\varepsilon}\right)$ for some $\varepsilon > 0$, $\exists c < 1.\ af\left(\frac{n}{b}\right) \leq cf(n)$,
   - ▷ then $T(n) = \Theta(f(n))$

Rough idea: does the pre/post-processing time $f(n)$ drive the complexity or the way the recursive calls handled?

Our examples

- Dichotomy/fast exponentiation: $a = 1, b = 2, f = \mathcal{O}(1)$

Intuitions for the master theorem

Master theorem $(T(n) = aT\left(\frac{n}{b}\right) + f(n))$

1. If $f(n) = \mathcal{O}(n^{\log_b(a)-\varepsilon})$ for some $\varepsilon > 0$,
   ▷ then $T(n) = \Theta(n^{\log_b(a)})$
2. If $f(n) = \Theta\left(n^{\log_b(a)} \log(n)^k\right)$ for some $k \geq 0$,
   ▷ then $T(n) = \Theta\left(n^{\log_b(a)} \log(n)^{k+1}\right)$
3. If $f(n) = \Omega\left(n^{\log_b(a)+\varepsilon}\right)$ for some $\varepsilon > 0$, $\exists c < 1.\ af\left(\frac{n}{b}\right) \leq cf(n)$,
   ▷ then $T(n) = \Theta(f(n))$

Rough idea: does the pre/post-processing time $f(n)$ drive the complexity or the way the recursive calls handled?

Our examples

- Dichotomy/fast exponentiation: $a = 1, b = 2, f = \mathcal{O}(1)$ Not covered: $\big(\log(n)\big)$

Intuitions for the master theorem

Master theorem $\left(T(n) = aT\left(\frac{n}{b}\right) + f(n)\right)$

1. If $f(n) = \mathcal{O}(n^{\log_b(a)-\varepsilon})$ for some $\varepsilon > 0$,
   $\triangleright$ then $T(n) = \Theta(n^{\log_b(a)})$
2. If $f(n) = \Theta\left(n^{\log_b(a)}\log(n)^k\right)$ for some $k \geq 0$,
   $\triangleright$ then $T(n) = \Theta\left(n^{\log_b(a)}\log(n)^{k+1}\right)$
3. If $f(n) = \Omega\left(n^{\log_b(a)+\varepsilon}\right)$ for some $\varepsilon > 0$, $\exists c < 1.\ af\left(\frac{n}{b}\right) \leq cf(n)$,
   $\triangleright$ then $T(n) = \Theta(f(n))$

Rough idea: does the pre/post-processing time $f(n)$ drive the complexity or the way the recursive calls handled?

Our examples

- Dichotomy/fast exponentiation: $a = 1$, $b = 2$, $f = \mathcal{O}(1)$ Not covered: $\left(\log(n)\right)$
- Majority:

Intuitions for the master theorem

Master theorem $\left(T(n) = aT\left(\frac{n}{b}\right) + f(n)\right)$

1. If $f(n) = \mathcal{O}(n^{\log_b(a)-\varepsilon})$ for some $\varepsilon > 0$,
   ▷ then $T(n) = \Theta(n^{\log_b(a)})$
2. If $f(n) = \Theta\left(n^{\log_b(a)} \log(n)^k\right)$ for some $k \geq 0$,
   ▷ then $T(n) = \Theta\left(n^{\log_b(a)} \log(n)^{k+1}\right)$
3. If $f(n) = \Omega\left(n^{\log_b(a)+\varepsilon}\right)$ for some $\varepsilon > 0$, $\exists c < 1.\ af\left(\frac{n}{b}\right) \leq cf(n)$,
   ▷ then $T(n) = \Theta(f(n))$

Rough idea: does the pre/post-processing time $f(n)$ drive the complexity or the way the recursive calls handled?

Our examples

- Dichotomy/fast exponentiation: $a = 1$, $b = 2$, $f = \mathcal{O}(1)$ Not covered: $\big(\log(n)\big)$
- Majority: $a = 2 = b$, $f = \mathcal{O}(n)$

Intuitions for the master theorem

Master theorem $(T(n) = aT\left(\frac{n}{b}\right) + f(n))$

1. If $f(n) = \mathcal{O}(n^{\log_b(a)-\varepsilon})$ for some $\varepsilon > 0$,
   - ▷ then $T(n) = \Theta(n^{\log_b(a)})$
2. If $f(n) = \Theta\left(n^{\log_b(a)} \log(n)^k\right)$ for some $k \geq 0$,
   - ▷ then $T(n) = \Theta\left(n^{\log_b(a)} \log(n)^{k+1}\right)$
3. If $f(n) = \Omega\left(n^{\log_b(a)+\varepsilon}\right)$ for some $\varepsilon > 0$, $\exists c < 1.\ af\left(\frac{n}{b}\right) \leq cf(n)$,
   - ▷ then $T(n) = \Theta(f(n))$

Rough idea: does the pre/post-processing time $f(n)$ drive the complexity or the way the recursive calls handled?

Our examples

- Dichotomy/fast exponentiation: $a = 1, b = 2, f = \mathcal{O}(1)$ Not covered: $\big(\log(n)\big)$
- Majority: $a = 2 = b, f = \mathcal{O}(n) \to 2. \to \mathcal{O}(n\log(n))$

Other paradigms

We have seen a few high-level ideas to develop efficient algorithms:

- try to generalize intuitive already available solutions you'd naturally execute on some examples
- think **recursively**: reduce solving an instance of size n to an instance of size $n-k$
- **divide and conquer**: reduce solving an instance of size n to solving instances of size $\frac{n}{k}$
- **dynamic programming**: cache common subcomputation across recursive calls

Other paradigms

We have seen a few high-level ideas to develop efficient algorithms:

- try to generalize intuitive already available solutions you'd naturally execute on some examples
- think **recursively**: reduce solving an instance of size n to an instance of size $n - k$
- **divide and conquer**: reduce solving an instance of size n to solving instances of size $\frac{n}{k}$
- **dynamic programming**: cache common subcomputation across recursive calls

Maybe one more today: **greedy algorithms**

A motivating example: the change problem

Problem

Input: Coin values $c_0, \ldots, c_n$ and an amount x

Output: A number of coins of each type $a_0, \ldots, a_k$ such that $\sum_i a_i c_i = x$

High-level considerations

- For these kind of optimization problems, dynamic programming typically gives optimal solutions in the most reasonable times
- But in many situations, a simple greedy algorithm might give optimal solutions!

High-level considerations

- For these kind of optimization problems, dynamic programming typically gives optimal solutions in the most reasonable times
- But in many situations, a simple greedy algorithm might give optimal solutions!

Optimized change problem

Input: Coin values $c_0, \ldots, c_n$ and an amount x

Output: A number of coins of each type $a_0, \ldots, a_k$ such that $\sum_i a_i c_i = x$ and $\sum_i a_i$ minimal amongs all possible solutions

High-level considerations

- For these kind of optimization problems, dynamic programming typically gives optimal solutions in the most reasonable times
- But in many situations, a simple greedy algorithm might give optimal solutions!

Optimized change problem

Input: Coin values $c_0, \ldots, c_n$ and an amount x

Output: A number of coins of each type $a_0, \ldots, a_k$ such that $\sum_i a_i c_i = x$ and $\sum_i a_i$ minimal amongs all possible solutions

Greedy algo: optimal if $2c_i \leq c_i + 1$ for all $i < n$!

Sorting algorithms

The problem

The sorting problem

Input: An array of integers of size n

Output: A sorted array containing the same elements

The problem

The sorting problem

Input: An array of integers of size n

Output: A sorted array containing the same elements

- For now, only arrays
- Later, fancier datastructures but essentially same asymptotic time
- **Motivation:** very classical problem and solutions, good case studies

The problem

The sorting problem

Input: An array of integers of size n

Output: A sorted array containing the same elements

- For now, only arrays
- Later, fancier datastructures but essentially same asymptotic time
- **Motivation:** very classical problem and solutions, good case studies

Last lab: bubble sort presented recursively! $\mathcal{O}(n^2)$

Insertion sort

Subproblem

Input: A sorted array of integers A of size n and element x

Output: A sorted array containing the same elements as A plus x

Insertion sort

Subproblem

Input: A sorted array of integers A of size n and element x

Output: A sorted array containing the same elements as A plus x

Can you write that?

Insertion sort

Subproblem

Input: A sorted array of integers A of size n and element x
Output: A sorted array containing the same elements as A plus x

Can you write that? What complexity?

Insertion sort

Subproblem

Input: A sorted array of integers A of size n and element x
Output: A sorted array containing the same elements as A plus x

Can you write that? What complexity? $\mathcal{O}(n)$

Insertion sort

Subproblem

Input: A sorted array of integers A of size n and element x

Output: A sorted array containing the same elements as A plus x

Can you write that? What complexity? $\mathcal{O}(n)$

Can you deduce a sorting algorithm?

Insertion sort

Subproblem

Input: A sorted array of integers A of size n and element x
Output: A sorted array containing the same elements as A plus x

Can you write that? What complexity? $\mathcal{O}(n)$

Can you deduce a sorting algorithm? What complexity?

Insertion sort

Subproblem

Input: A sorted array of integers A of size n and element x
Output: A sorted array containing the same elements as A plus x

Can you write that? What complexity? $\mathcal{O}(n)$

Can you deduce a sorting algorithm? What complexity? $\mathcal{O}(n^2)$

Merge sort

Can you think of a divide-and-conquer approach?

Merge sort

Can you think of a divide-and-conquer approach?

Idea

- Split the array into two equal pieces
- Sort the two pieces recursively
- *Merge* the two pieces back together

Merging two arrays

Subproblem

Input: Two sorted arrays of integers A and B

Output: A sorted array containing the same elements as A plus B

Merging two arrays

Subproblem

Input: Two sorted arrays of integers A and B

Output: A sorted array containing the same elements as A plus B

Complexity?

Merging two arrays

Subproblem

Input: Two sorted arrays of integers A and B

Output: A sorted array containing the same elements as A plus B

Complexity? $\mathcal{O}(n)$

Merge sort's complexity

- Splitting the arrays: $\mathcal{O}(n)$ naively, $\mathcal{O}(1)$ with some mild alteration to the inputs

  (use arrays + sorted ranges as inputs rather than arrays and work *in-place*)
- Merging things together: $\mathcal{O}(n)$

Complexity?

Merge sort's complexity

- Splitting the arrays: $\mathcal{O}(n)$ naively, $\mathcal{O}(1)$ with some mild alteration to the inputs
  (use arrays + sorted ranges as inputs rather than arrays and work *in-place*)
- Merging things together: $\mathcal{O}(n)$

Complexity? $\rightarrow$ Master theorem $\rightarrow$

Merge sort's complexity

- Splitting the arrays: $\mathcal{O}(n)$ naively, $\mathcal{O}(1)$ with some mild alteration to the inputs
  (use arrays + sorted ranges as inputs rather than arrays and work *in-place*)
- Merging things together: $\mathcal{O}(n)$

Complexity? $\to$ Master theorem $\to$ $\mathcal{O}(n \log(n))$

Quick sort

Idea: instead of making the splitting trivial, make the merging trivial

- Pick an element, the *pivot*
- Write two subarrays of elements: those smaller than the pivot, and those larger
- Sort recursively and concatenate the results

Quick sort's complexity

- **Worst case:** $\mathcal{O}(n^2)$ for a bad choice of pivot
- **Best case:** $\mathcal{O}(n\log(n))$ for a good choice (the median) (or if lucky)

(A median can be picked in linear time actually)

(but a lot of implementations don't bother)

(it's a *fancy* divide-and-conquer algo)

- **Average case:** $\mathcal{O}(n\log(n))$

Actually $\mathcal{O}(n \log(n))$ is optimal

But is it? (sorting by counting)

Advanced considerations: sorting in-place, stable sorts, parallelism

Reading suggestions

- The background reading here $\rightsquigarrow$ go more in-depth with the material

(you don't *need* to read all of that immediately)

Algorithms in Java (3rd ed., 2004) by Sedgewick

Relevant chapters: 6,7,8 and 10

Explain and study sorting algorithms in details

Introduction to Algorithms (4th ed., 2011) by Cormen et. al

Relevant chapters: 4,7,8,14,15

More focus on paradigms

What now?

- Practice! Both coming up with algorithms and implementation
- You've had roughly a quick overview of the main points an undergrad first algorithmics module would cover
- The first CW will be over this material.
- Next up: datastuctures!
  - Algorithms for and with datastructures!

What now?

- Practice! Both coming up with algorithms and implementation
- You've had roughly a quick overview of the main points an undergrad first algorithmics module would cover
- The first CW will be over this material.
- Next up: datastuctures!
    - Algorithms for and with datastructures!

OK, time for questions?