CSCM12: software concept and efficiency
Estimating the complexity of algorithms

Cécilia Pradic

Swansea University, 06/20/2023

Recommended reading after this lecture

- **Chapter 3** "Characterizing Running Times"
  of *Introduction to Algorithms* (4th ed., 2011) by Cormen et. al
- **Chapter 2** "Principles of Algorithm Analysis"
  of *Algorithms in Java* (3rd ed., 2004) by Sedgewick

  No need to look at the "Basic Recurrences" section for now

One running example

An algorithmic problem

Input: An array A of size n and some (say, integer) x

Output: An index i such that $A[i] = x$ or -1 if there is none

Solution #1

`FindIndex`(A, x)

$res \leftarrow -1$
$n \leftarrow$ size of A
for i **from** 0 **to** $n - 1$ **do**
    if $A[i] = x$ **then**
        $res \leftarrow i$
return res

Running the first solution

Let us try to run this step-by-step!

FindIndex(A, x)

$res \leftarrow -1$
$n \leftarrow$ size of A
for i **from** 0 **to** $n-1$ **do**
    if $A[i] = x$ **then**
        $res \leftarrow i$
return res

- $A = [2, 4, 7, 7, 10, 15], x = 7$

Running the first solution

Let us try to run this step-by-step!

```
FindIndex(A, x)
    res ← −1
    n ← size of A
    for i from 0 to n − 1 do
        if A[i] = x then
            res ← i
    return res
```

- $A = [2, 4, 7, 7, 10, 15], x = 7$
- $A = [2, 4, 7, 7, 10, 15], x = 11$

Alternative solution 1

Solution #2

```
FindIndex2(A, x)
    res ← −1
    n ← size of A
    for i from n − 1 down to 0 do
        if A[i] = x then
            res ← i
    return res
```

- Solves the same problem
- Different outputs on our first sample input
- (Roughly the same complexity)

Alternative solution 2

Solution #3

```
FindIndex3(A, x)
    res ← −1
    n ← size of A
    i ← 0
    while res = −1 and i < n do
        if A[i] = x then
            res ← i
        Increment i
    return res
```

- Sometimes more efficient
- But is it significant in practice?

A more precise problem and another solution

A more precise algorithmic problem

Input: A **sorted** array A of size n and some (say, integer) x

Output: An index i such that $A[i] = x$ or -1 if there is none

- The previous solutions work, but...

A more efficient solution for sorted inputs

```
FindIndexDicho(A, x)
    start ← 0
    end ← size of A
    while start < end do
        mid ← ⌈(end+start)/2⌉
        if A[mid] ≤ x then
            start ← mid
        else
            end ← mid

    if A[start] = x then
        return start
    else
        return -1
```

Consideration of efficiency

Given an algorithmic problem:

- Is there an algorithm that solves it? If so is it:
  - feasible? (usable in practice)
  - efficient?
  - optimal?

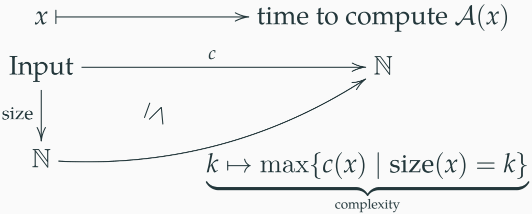
Given an algorithm:

- How efficient is it?
- Is there a more method of getting the same results?

Rules of thumb for measuring efficiency

- *Typically*, (time) complexity mostly depends on the **size** of the input

$\to$ we typically express the time complexity as a function "size $\mapsto$ time"



Note the $\leq$: typically we want the **worst-case complexity** for inputs of a given size

- best-case: not very interesting
- average: can be interesting, typically harder to compute though :)

Computing time complexity

- Can be roughly be done step-by-step.
- Essentially, each piece of a program can be regarded as a mathematical function

$$\overbrace{\text{State}}^{\text{(initial) value of variables/memory}} \longrightarrow \text{State} \times \underbrace{\mathbb{N}}_{\text{time taken to compute the step}}$$

- Essentially: basic arithmetic operations, assignments: cost ~ 1, array allocation $\sim$ size of the array, loop $\sim$ sum of the complexities, ...

$\rightarrow$ roughly the number of steps in step-wise execution we've done

The notion of space complexity

There is a notion of **space** complexity

- Essentially, assign a size to State and compute the maximal size that occurs in an execution
- Unless you are doing big data or embedded system, this is not typically a limiting factor

(RAM is cheap)

- In most scenarii, bounded by time complexity

Accurate complexity?

The "time complexity function" we defined might not be completely accurate

In practice

- hardware/compiler-dependent behaviors
- not so reliable hardware optimisations

(predictive branching, prefetch, cache misses)

→ We had to make compromises

Accurate complexity?

The "time complexity function" we defined might not be completely accurate

In practice

- hardware/compiler-dependent behaviors
- not so reliable hardware optimisations

(predictive branching, prefetch, cache misses)

→ We had to make compromises

However, gives reasonable bounds/estimate

- up to a **constant factor**
- for **large inputs** (and that's we care about!)

Functions at infinity up to a constant

Suppose that we have two complexity functions $f, g : \mathbb{N}_{>0} \to \mathbb{R}^+$

- Say that g *asymptotically dominates* f if $f \leq K \cdot g + K'$ for some $K, K' > 0$

(vocabulary: asymptotically = "at the limit")

Functions at infinity up to a constant

Suppose that we have two complexity functions $f, g : \mathbb{N}_{>0} \to \mathbb{R}^+$

- Say that g *asymptotically dominates* f if $f \leq K \cdot g + K'$ for some $K, K' > 0$

(vocabulary: asymptotically = "at the limit")

$$\longrightarrow \text{ Try to compute } \lim_{n \to +\infty} \frac{f(n)}{g(n)}$$

Functions at infinity up to a constant

Suppose that we have two complexity functions $f, g : \mathbb{N}_{>0} \to \mathbb{R}^+$

- Say that g *asymptotically dominates* f if $f \leq K \cdot g + K'$ for some $K, K' > 0$

(vocabulary: asymptotically = "at the limit")

$$\longrightarrow \text{ Try to compute } \lim_{n \to +\infty} \frac{f(n)}{g(n)}$$

- if that's finite and non-zero: f and g are commensurate

Functions at infinity up to a constant

Suppose that we have two complexity functions $f, g : \mathbb{N}_{>0} \to \mathbb{R}^+$

- Say that g *asymptotically dominates* f if $f \leq K \cdot g + K'$ for some $K, K' > 0$

(vocabulary: asymptotically = "at the limit")

$$\longrightarrow \text{ Try to compute } \lim_{n \to +\infty} \frac{f(n)}{g(n)}$$

- if that's finite and non-zero: f and g are commensurate
- if that's $+\infty$: f dominates strictly g asymptotically

Functions at infinity up to a constant

Suppose that we have two complexity functions $f, g : \mathbb{N}_{>0} \to \mathbb{R}^+$

- Say that g *asymptotically dominates* f if $f \leq K \cdot g + K'$ for some $K, K' > 0$

(vocabulary: asymptotically = "at the limit")

$$\longrightarrow \text{ Try to compute } \lim_{n \to +\infty} \frac{f(n)}{g(n)}$$

- if that's finite and non-zero: f and g are commensurate
- if that's $+\infty$: f dominates strictly g asymptotically
- if that's 0: g dominates f strictly asymptotically

Big $\mathcal{O}$ notation and friends

Very important notations

- $f(n) = \mathcal{O}(g(n))$ means $\lim_{n \to +\infty} \frac{f(n)}{g(n)} < +\infty$ **That's the one you'll see all the time**

Big $\mathcal{O}$ notation and friends

Very important notations

- $f(n) = \mathcal{O}(g(n))$ means $\lim_{n \to +\infty} \frac{f(n)}{g(n)} < +\infty$ **That's the one you'll see all the time**
- $f(n) = \Omega(g(n))$ means $0 < \lim_{n \to +\infty} \frac{f(n)}{g(n)}$

Very important notations

- $f(n) = \mathcal{O}(g(n))$ means $\lim_{n \to +\infty} \frac{f(n)}{g(n)} < +\infty$ **That's the one you'll see all the time**
- $f(n) = \Omega(g(n))$ means $0 < \lim_{n \to +\infty} \frac{f(n)}{g(n)}$
- $f(n) = \Theta(g(n))$ means $f(n) = \mathcal{O}(g(n))$ and $f(n) = \Omega(g(n))$

Big $\mathcal{O}$ notation and friends

Very important notations

- $f(n) = \mathcal{O}(g(n))$ means $\lim_{n \to +\infty} \frac{f(n)}{g(n)} < +\infty$ **That's the one you'll see all the time**
- $f(n) = \Omega(g(n))$ means $0 < \lim_{n \to +\infty} \frac{f(n)}{g(n)}$
- $f(n) = \Theta(g(n))$ means $f(n) = \mathcal{O}(g(n))$ and $f(n) = \Omega(g(n))$
- $f(n) = o(g(n))$ means $\lim_{n \to +\infty} \frac{f(n)}{g(n)} = 0$

Big $\mathcal{O}$ notation and friends

Very important notations

- $f(n) = \mathcal{O}(g(n))$ means $\lim_{n \to +\infty} \frac{f(n)}{g(n)} < +\infty$ **That's the one you'll see all the time**
- $f(n) = \Omega(g(n))$ means $0 < \lim_{n \to +\infty} \frac{f(n)}{g(n)}$
- $f(n) = \Theta(g(n))$ means $f(n) = \mathcal{O}(g(n))$ and $f(n) = \Omega(g(n))$
- $f(n) = o(g(n))$ means $\lim_{n \to +\infty} \frac{f(n)}{g(n)} = 0$

Basic examples:

- $n = \mathcal{O}(n^2)$
- $n^3 + n^2 + \log(n) = \Theta(5n^3)$
- $\log(n)2^n + n^5 + 5 = \Theta(\log(n)2^n)$
- $\log(n) = o(\sqrt{n})$
- $42 + \frac{1}{n} = \mathcal{O}(1)$

Basic tips for computing with $\mathcal{O}$

- If $f(n) \leq g(n)$ then $f(n) = \mathcal{O}(g(n))$
- $f(n) = o(g(n))$ impies $f(n) = \mathcal{O}(g(n))$
- for any $k > 0$ and k', $kf(n) = \mathcal{O}(f(n))$
- If $f(n) = \mathcal{O}(g(n))$ and $g(n) = \mathcal{O}(h(n))$ then $f(n) = \mathcal{O}(h(n))$
- $\log(n)^k = o(n)$, $n^k = o(2^n)$ for any constant $k \in \mathbb{R}^+$

  $k = \frac{1}{2}$ corresponds to $\sqrt{}$
- $n^k = o(n^{k'})$ for $k < k'$
- $f_1(n) = \mathcal{O}(g_1(n))$ and $f_2 = \mathcal{O}(g_2(n))$ imply $f_1(n)f_2(n) = \mathcal{O}(g_1(n)g_2(n))$
- If $f(n) = o(g(n))$, then $f(n) + g(n) = \mathcal{O}(f(n))$

Back to our examples (1/4)

Solution #1

```
FindIndex(A, x)
    res ← −1
    n ← size of A
    for i from 0 to n − 1 do
        if A[i] = x then
            res ← i
    return res
```

Worst-case complexity?

Solution #1

```
FindIndex(A, x)
    res ← −1
    n ← size of A
    for i from 0 to n − 1 do
        if A[i] = x then
            res ← i
    return res
```

Worst-case complexity? $\rightarrow \mathcal{O}(n)$ (linear)

Back to our examples (1/4)

Solution #1

```
FindIndex(A, x)
    res ← −1
    n ← size of A
    for i from 0 to n − 1 do
        if A[i] = x then
            res ← i
    return res
```

Worst-case complexity? $\rightarrow \mathcal{O}(n)$ (linear)

worst-case	best-case	average case
$\Theta(n)$	$\Theta(n)$	$\Theta(n)$

Useful heuristics

- If/then/else $\rightsquigarrow$ can be over-approximated by the max of each branch
- Loops: if the body runs in $\mathcal{O}(f(n))$ and there are $\mathcal{O}(g(n))$ iterations $\rightarrow \mathcal{O}(f(n)g(n))$

Back to our examples (2/4)

Solution #2

```
FindIndex2(A, x)
    res ← −1
    n ← size of A
    for i from n − 1 down to 0 do
        if A[i] = x then
            res ← i
    return res
```

Worst-case complexity?

Back to our examples (2/4)

Solution #2

```
FindIndex2(A, x)
    res ← −1
    n ← size of A
    for i from n − 1 down to 0 do
        if A[i] = x then
            res ← i
    return res
```

Worst-case complexity? → $\mathcal{O}(n)$ (nothing so different)

worst-case	best-case	average case
$\Theta(n)$	$\Theta(n)$	$\Theta(n)$

Solution #3

```
FindIndex3(A, x)
    res ← −1
    n ← size of A
    i ← 0
    while res = −1 and i < n do
        if A[i] = x then
            res ← i
        Increment i
    return res
```

Worst-case complexity? $\rightarrow \mathcal{O}(n)$ (nothing too different)

Back to our examples (3/4)

Solution #3

```
FindIndex3(A, x)
    res ← −1
    n ← size of A
    i ← 0
    while res = −1 and i < n do
        if A[i] = x then
            res ← i
        Increment i
    return res
```

Worst-case complexity? $\rightarrow \mathcal{O}(n)$ (nothing too different)

But...

worst-case	best-case	average case
$\Theta(n)$	$\Theta(1)$	$\Theta(n)$

Back to our examples (4/4)

(Recall that this one only works for *sorted* inputs)

```
FindIndexDicho(A, x)
    start ← 0
    end ← size of A
    while start < end do
        mid ← ⌈(end+start)/2⌉
        if A[mid] ≤ x then
            start ← mid
        else
            end ← mid

    if A[start] = x then
        return start
    else
        return -1
```

- Difficulty: number of iterations?

Back to our examples (4/4)

(Recall that this one only works for *sorted* inputs)

```
FindIndexDicho(A, x)
    start ← 0
    end ← size of A
    while start < end do
        mid ← ⌈(end+start)/2⌉
        if A[mid] ≤ x then
            start ← mid
        else
            end ← mid

    if A[start] = x then
        return start
    else
        return -1
```

- Difficulty: number of iterations?
- At step k, $end - start \leq \left\lfloor \frac{n}{2^k} \right\rfloor$

Back to our examples (4/4)

(Recall that this one only works for *sorted* inputs)

```
FindIndexDicho(A, x)
    start ← 0
    end ← size of A
    while start < end do
        mid ← ⌈(end+start)/2⌉
        if A[mid] ≤ x then
            start ← mid
        else
            end ← mid

    if A[start] = x then
        return start
    else
        return -1
```

- Difficulty: number of iterations?
- At step k, $end - start \leq \left\lfloor \frac{n}{2^k} \right\rfloor$
- Main loop ends when $start = end$

Back to our examples (4/4)

(Recall that this one only works for *sorted* inputs)

```
FindIndexDicho(A, x)
    start ← 0
    end ← size of A
    while start < end do
        mid ← ⌈(end+start)/2⌉
        if A[mid] ≤ x then
            start ← mid
        else
            end ← mid

    if A[start] = x then
        return start
    else
        return -1
```

- Difficulty: number of iterations?
- At step k, $end - start \leq \left\lfloor \frac{n}{2^k} \right\rfloor$
- Main loop ends when $start = end$

$\rightarrow$ when $\frac{n}{2^k} < 1$

Back to our examples (4/4)

(Recall that this one only works for *sorted* inputs)

FindIndexDicho(A, x)

- $start \leftarrow 0$
- $end \leftarrow$ size of A
- **while** $start < end$ **do**
  - $mid \leftarrow \lceil \frac{end+start}{2} \rceil$
  - **if** $A[mid] \leq x$ **then**
    - $start \leftarrow mid$
  - **else**
    - $end \leftarrow mid$
- **if** $A[start] = x$ **then**
  - **return** $start$
- **else**
  - **return** *-1*

- Difficulty: number of iterations?
- At step k, $end - start \leq \left\lfloor \frac{n}{2^k} \right\rfloor$
- Main loop ends when $start = end$

→ when $\frac{n}{2^k} < 1$

→ when $n < 2^k$

Back to our examples (4/4)

(Recall that this one only works for *sorted* inputs)

FindIndexDicho(A, x)

- $start \leftarrow 0$
- $end \leftarrow$ size of A
- **while** $start < end$ **do**
  - $mid \leftarrow \left\lceil \frac{end+start}{2} \right\rceil$
  - **if** $A[mid] \leq x$ **then**
    - $start \leftarrow mid$
  - **else**
    - $end \leftarrow mid$
- **if** $A[start] = x$ **then**
  - **return** $start$
- **else**
  - **return** *-1*

Complexity?

- Difficulty: number of iterations?
- At step k, $end - start \leq \left\lfloor \frac{n}{2^k} \right\rfloor$
- Main loop ends when $start = end$

$\rightarrow$ when $\frac{n}{2^k} < 1$

$\rightarrow$ when $n < 2^k$

$\rightarrow$ when $\log_2(n) < k$

Back to our examples (4/4)

(Recall that this one only works for *sorted* inputs)

```
FindIndexDicho(A, x)
    start ← 0
    end ← size of A
    while start < end do
        mid ← ⌈(end+start)/2⌉
        if A[mid] ≤ x then
            start ← mid
        else
            end ← mid

    if A[start] = x then
        return start
    else
        return -1
```

Complexity? $\rightarrow \Theta(\log(n))$

- Difficulty: number of iterations?
- At step k, $end - start \leq \left\lfloor \frac{n}{2^k} \right\rfloor$
- Main loop ends when $start = end$

$\rightarrow$ when $\frac{n}{2^k} < 1$

$\rightarrow$ when $n < 2^k$

$\rightarrow$ when $\log_2(n) < k$

Some simple examples (1/3)

```
SumTensor(A)
    n ← size of A
    r ← 0
    for i from n − 1 down to 0 do
        for j from 0 to n − 1 do
            r ← A[i] × A[j]
    return r
```

Complexity?

Some simple examples (1/3)

```
SumTensor(A)
    n ← size of A
    r ← 0
    for i from n − 1 down to 0 do
        for j from 0 to n − 1 do
            r ← A[i] × A[j]
    return r
```

Complexity? → $\Theta(n^2)$ (quadratic)

Some simple examples (2/3)

```
SumLowerTensor(A)
    n ← size of A
    r ← 0
    for i from n − 1 down to 0 do
        for j from 0 to i do
            r ← A[i] × A[j]
    return r
```

Complexity?

Some simple examples (2/3)

```
SumLowerTensor(A)
    n ← size of A
    r ← 0
    for i from n − 1 down to 0 do
        for j from 0 to i do
            r ← A[i] × A[j]
    return r
```

Complexity? $\rightarrow \mathcal{O}(n^2)$

Some simple examples (2/3)

```
SumLowerTensor(A)
    n ← size of A
    r ← 0
    for i from n − 1 down to 0 do
        for j from 0 to i do
            r ← A[i] × A[j]
    return r
```

Complexity? $\rightarrow \mathcal{O}(n^2)$ in fact $\Theta(n^2)$

Some simple examples (2/3)

```
SumLowerTensor(A)
    n ← size of A
    r ← 0
    for i from n − 1 down to 0 do
        for j from 0 to i do
            r ← A[i] × A[j]
    return r
```

Complexity? $\rightarrow \mathcal{O}(n^2)$ in fact $\Theta(n^2)$

Lower bound: $\sum_{i=0}^{n} i = \frac{n(n+1)}{2} = \Theta(n^2)$

Some simple examples (2/3)

```
SumLowerTensor(A)
    n ← size of A
    r ← 0
    for i from n − 1 down to 0 do
        for j from 0 to i do
            r ← A[i] × A[j]
    return r
```

Complexity? $\rightarrow \mathcal{O}(n^2)$ in fact $\Theta(n^2)$

Lower bound: $\sum_{i=0}^{n} i = \frac{n(n+1)}{2} = \Theta(n^2)$

(more generally, $\sum_{i=0}^{n} i^k = \Theta(n^k)$, so that kind of approximation is often safe)

Some simple examples (3/3)

Recall that SumTensor is $\mathcal{O}(n^2)$

```
Something weird(A)
    n ← size of A
    r ← 0
    for i from n − 1 down to 0 do
        r ← A[i%2] × SumTensor(A)
    return r
```

Complexity?

Some simple examples (3/3)

Recall that SumTensor is $\mathcal{O}(n^2)$

```
Something weird(A)
    n ← size of A
    r ← 0
    for i from n − 1 down to 0 do
        r ← A[i%2] × SumTensor(A)
    return r
```

Complexity? $\rightarrow \mathcal{O}(n^3)$

Complexity of an algorithmic problem

- Recall that an algorithmic problem $\neq$ algorithm.
- Common shorthands for the intrinsic hardness of a problem **P**:
  - **P** is in $\mathcal{O}(f(n)) \rightarrow$ there is a $\mathcal{O}(f(n))$ algorithm solving **P**
  - **P** is in $\Theta(f(n)) \rightarrow$ there is an optimal solution to **P** in $\Theta(f(n))$
  - **P** is in $\Omega(f(n)) \rightarrow$ any algorithm solving **P** has complexity $\Omega(f(n))$

(out of scope) complexity theory

Are some problem intrinsically hard $\rightarrow$ yes!

- Complexity theorists study that!
- Problems solvable in $\mathcal{O}(n^k)$ = solvable in polynomial time, class P
- Problems whose solution can be checked in polynomial time NP

Typically

- Polynomial time problems are tractable
- Problems that are NP-hard do not have known subexponential solution

$\rightarrow$ to prove that some problem is intricically hard, prove it is necessarily as hard as *all* NP problems

Big open problem

Is P $\neq$ NP?

(there are classes that are strictly harder than NP, such as EXPTIME)

Next challenge to compute complexities

```
FindIndexDicho2(A, x, start, end)
    if end ≤ start then
        if A[start] = x then
            return start
        else
            return -1
    mid ← ⌈(end+start)/2⌉
    if A[mid] ≤ x then
        FindIndexDicho2(A, x, mid, end)
    else
        FindIndexDicho2(A, x, start, mid)
```

$C(0) = \mathcal{O}(1)$

$C(n+1) = C\left(\left\lfloor \frac{n+1}{2} \right\rfloor\right) + \mathcal{O}(1)$

Next challenge to compute complexities

```
FindIndexDicho2(A, x, start, end)
    if end ≤ start then
        if A[start] = x then
            return start
        else
            return -1
    mid ← ⌈(end+start)/2⌉
    if A[mid] ≤ x then
        FindIndexDicho2(A, x, mid, end)
    else
        FindIndexDicho2(A, x, start, mid)
```

$C(0) = \mathcal{O}(1)$

$C(n+1) = C\left(\left\lfloor \frac{n+1}{2} \right\rfloor\right) + \mathcal{O}(1)$

$\rightarrow C(n) = \mathcal{O}(\log(n))$

Conclusion

Thanks for listening!

Please look at the resources on canvas as well

Strike & logistics

Questions?