Asymptotic Analysis & Sorting

The Fast and the Recursiveness

15-150 M21

Lecture 0609
09 June 2021

0 Sequential Runtime Analysis

Today's slogan

Think big (and think long-term)

Big-O as a vocabulary for expressing asymptotic runtime

`f` is $O(1)$ - doubling the input size to `f` doesn't change the runtime

`f` is $O(n)$ - doubling the input size to `f` doubles the runtime

`f` is $O(\log n)$ - doubling the input size to `f` adds a constant to the runtim

`f` is $O(n^2)$ - doubling the input size to `f` quadruples the runtime

Check Your Understanding

What Big-O class is `pow` in?

A $O(1)$
B $O(\log n)$
C $O(n)$
D $O(n^2)$

Asymptotic Standpoint solves a few issues...

- Internal representations?

$$\texttt{6+6} \Longrightarrow^{?k} \texttt{12} \qquad \text{(for some constant } k\text{)}$$

- Hardware dependence: want algorithm analysis to be the same for both our computers, even if yours is twice as fast as mine. (constant coefficients are "consumed" by the Big-O)
- For small inputs, the costs of initiating the process and storing variables is much higher relative to the costs of actual computation (assuming asymptotically huge inputs, so not a problem)

Want: General method to determine which Big-O class `f` is in

The Tree Method

0 How you're quantifying input size
1 Recurrence
2 Description of work tree
3 Measurements of work tree (height, and width at each level)
4 Summation
5 Big-O

Worked example: @ analysis

0609.0 (work.sml)

```
fun [] @ L2 = L2
  | (x::xs) @ L2 = x::(xs@L2)
```

0 Measure of input size: length of the first list

1 Recurrence

Big Idea

Abstractly define $W : \mathbb{N} \to \mathbb{N}$ such that evaluating `f(x)` for an input of size n takes approximately $W(n)$ steps, then classify the Big-O complexity of W

0609.0 (work.sml)

```
fun [] @ L2 = L2
  | (x::xs) @ L2 = x::(xs@L2)
```

0 Measure of input size: length of the first list

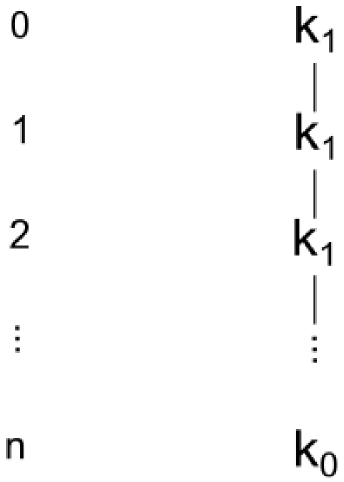
1 Recurrence:

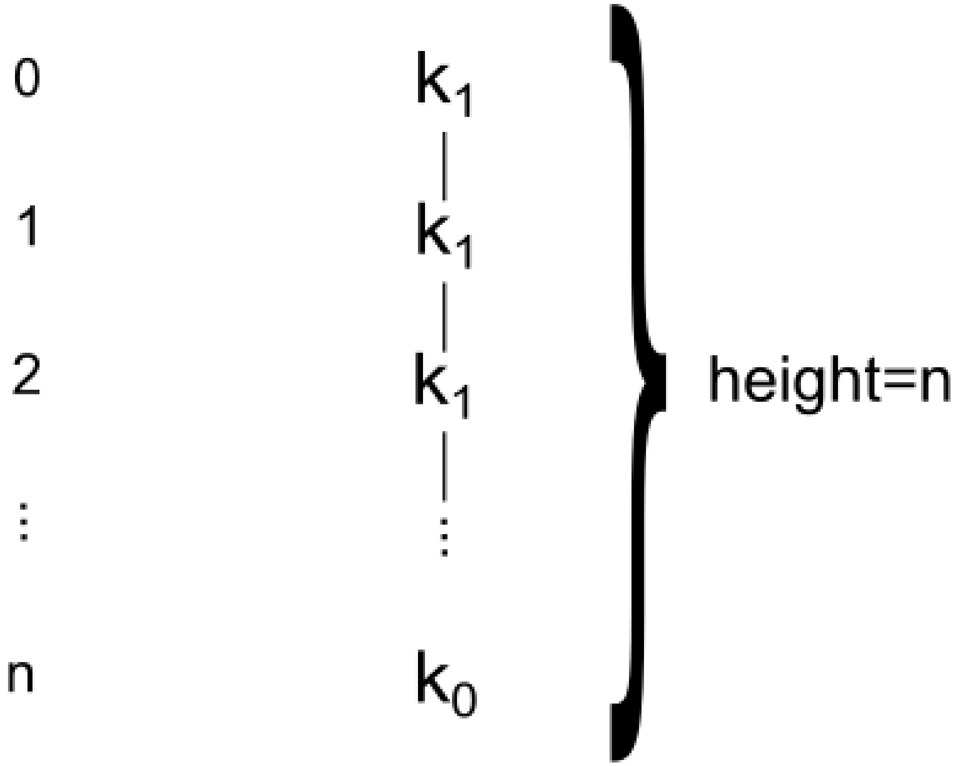
$$\begin{aligned} W(0) &= k_0 \\ W(n) &= k_1 + W(n-1) \end{aligned}$$

1. Recurrence:

$$W(0) = k_0$$
$$W(n) = k_1 + W(n-1)$$

2. Work Tree




0
1
2
⋮
n
k_1
k_1
k_1
⋮
k_0
height=n

3. Measurements
   Height: n
   Work on the i-th level: k_1
4. Sum:

$$\begin{aligned} W(n) &= k_0 + \sum_{i=0}^{n-1} k_1 \\ &\approx n k_1 \end{aligned}$$

5. Big O:

$$W(n) \text{ is } O(n)$$

0609.0 (work.sml)

```
(* op@ : t list * t list -> t list
 * REQUIRES: true
 * ENSURES: L1@L2 is the list consisting
 *          of the elements of L1 (in order),
 *          followed by the elements of L2
 *          (in order)
 * WORK: O(n), where n = |L1|
 *)
fun [] @ L2 = L2
  | (x::xs) @ L2 = x::(xs@L2)
```

Worked example: `pow` analysis

Built-in arithmetic is assumed to be constant-time

0609.1 (work.sml)

```
(* O(1) *)
fun square (x:int):int = x * x

(* O(1) *)
fun even (n : int):bool = (n mod 2)=0
```

New pow

0609.2 (work.sml)

```
fun pow 0 = 1
  | pow n =
      case (even n) of
        true  => square(pow(n div 2))
      | false => 2*square(pow(n div 2))
```

New pow

0609.2 (work.sml)

```
(* pow : int -> int
 * REQUIRES: n>=0
 * ENSURES: pow(n) == exp(n)
 * WORK: O(log n)
 *)
fun pow 0 = 1
  | pow n =
      case (even n) of
        true  => square(pow(n div 2))
      | false => 2*square(pow(n div 2))
```

Check Your Understanding

Determine the Big-O complexity of `exp`

Worked example: `rev` analysis

0609.3 (work.sml)

```
(* rev : t list -> t list
 * REQUIRES: true
 * ENSURES: (rev L) == L', where L' is the list
 *           containing the same elements as L,
 *           but in the opposite order
 * WORK: O(n^2), where n = |L|
 *)
fun rev [] = []
  | rev (x::xs) = (rev xs)@[x]
```

Check Your Understanding

Determine the Big-O complexity of `trev`

5-minute break

1 Sorting

Order

SML has a built-in type called `order`. It has three constructors/values:

`LESS` `EQUAL` `GREATER`

```
Int.compare : int * int -> order
String.compare : string * string -> order
```

```
fun quadrantV1 (m:int,n:int):string =
  if m=0 orelse n=0
  then "boundary"
  else if m>0
       then if n>0
            then "I"
            else "IV"
       else if n<0
            then "II"
            else "III"
```

```
fun quadrant (m:int,n:int):string =
  case (Int.compare(m,0),Int.compare(n,0)) of
       (EQUAL, _ ) => "boundary"
    |  ( _ , EQUAL) => "boundary"
    |  (GREATER,GREATER) => "I"
    |  (LESS,GREATER) => "II"
    |  (LESS,LESS) => "III"
    |  (GREATER,LESS) => "IV"
```

Sorting

Sorting is a classic algorithmic problem in computer science: finding the fastest way to put all the elements of a list in order.

A value `[x_1,...,x_n] : int list` is **sorted** if for each $i = 1, \ldots, n-1$, `Int.compare(x_i,x_(i+1))` $\ncong$ `GREATER`.

Or, recursively: a value `v : int list` is **sorted** if either `v=[]` or `v=[x]` for some `x`, or `v=x::x'::xs` where `Int.compare(x,x')` $\ncong$ `GREATER` and `x'::xs` is sorted.

Spec

0609.4 (sorting.sml)

```
fun isSorted ([]:int list):bool = true
  | isSorted [x] = true
  | isSorted (x::x'::xs) =
      (x<=x') andalso isSorted(x'::xs)
```

`sort : int list -> int list`
REQUIRES: true
ENSURES: `sort(L)` evaluates to a sorted permutation of `L`

A "permutation" of `L` is just a list that contains the same elements the same number of times as `L`, just in a possibly different order. So `[1,1,2,3]` is a permutation of `[3,1,2,1]` but not of `[3,2,1]`.

Sorting Algorithms

There are many sorting algorithms: insertion sort, quick sort, merge sort, bubble sort, . . .

We'll be focusing on *merge sort*, which consists of the following three steps:

1. Split the input list in half
2. Sort each half
3. *merge* the sorted halves together to obtain a sorted whole

Specs

`split : int list -> int list * int list`
REQUIRES: true
ENSURES: `split L` evaluates to `(A,B)` where `A` and `B` differ in length by at most one, and `A@B` is a permutation of `L`

`merge : int list * int list -> int list`
REQUIRES: `A` and `B` are sorted
ENSURES: `merge(A,B)` evaluates to a sorted permutation of `A@B`

`msort : int list -> int list`
REQUIRES: true
ENSURES: `msort(L)` evaluates to a sorted permutation of `L`

`split : int list -> int list * int list`
REQUIRES: true
ENSURES: `split L` evaluates to `(A,B)` where `A` and `B` differ in length by at most one, and `A@B` is a permutation of `L`

0609.5 (sorting.sml)

```
fun split ([]):int list * int list = ([],[])
  | split ([x] : int list) = ([x],[])
  | split (x::x'::xs) =
    let
       val (A,B) = split xs
    in
      (x::A,x'::B)
    end
```

`merge : int list * int list -> int list`

REQUIRES: `A` and `B` are sorted

ENSURES: `merge(A,B)` evaluates to a sorted permutation of `A@B`

0609.6 (sorting.sml)

```
fun merge (L1:int list,[]:int list) = L1
  | merge ([],L2) = L2
  | merge (x::xs,y::ys) =
      (case Int.compare(x,y) of
         GREATER => y::merge(x::xs,ys)
       | _ => x::merge(xs,y::ys))
```

`msort : int list -> int list`

REQUIRES: true

ENSURES: `msort(L)` evaluates to a sorted permutation of `L`

0609.7 (sorting.sml)

```
fun msort([]:int list):int list = []
  | msort [x] = [x]
  | msort L =
      let
         val (A,B) = split L
      in
        merge(msort A,msort B)
      end
```

Analysis

0609.5 (sorting.sml)

```
fun split ([]):int list * int list = ([],[])
  | split ([x] : int list) = ([x],[])
  | split (x::x'::xs) =
    let
       val (A,B) = split xs
    in
      (x::A,x'::B)
    end
```

0 Measure of size: length of input list

1-4 . . .

5 $W_{\texttt{split}}(n)$ is $O(n)$

0609.6 (sorting.sml)

```
fun merge (L1:int list,[]:int list) = L1
  | merge ([],L2) = L2
  | merge (x::xs,y::ys) =
     (case Int.compare(x,y) of
        GREATER => y::merge(x::xs,ys)
      | _ => x::merge(xs,y::ys))
```

0 Measure of size: sum of lengths of input lists

1-4 . . .

5 $W_{\texttt{merge}}(n)$ is $O(n)$

0609.7 (sorting.sml)

```
fun msort([]:int list):int list = []
  | msort [x] = [x]
  | msort L =
    let
       val (A,B) = split L
    in
      merge(msort A,msort B)
    end
```

0 Measure of size: length of input list

1

$$
\begin{aligned}
W_{\texttt{msort}}(0) &= k_0 \\
W_{\texttt{msort}}(1) &= k_1 \\
W_{\texttt{msort}}(n) &\le k_2 + k_3 n + W_{\texttt{msort}}(n/2) + W_{\texttt{msort}}(n/2) + k_4 n \\
&\approx 2W_{\texttt{msort}}(n/2) + kn
\end{aligned}
$$

2 . . .

3 Height: $\log n$, work on level i: $k(n - i)$

4 . . .

5 $W_{\texttt{msort}}(n)$ is $O(n \log n)$

Summary

- We can make use of the formalism of recurrences and asymptotic complexity to precisely articulate the runtime of recursive functional functions/algorithms
- The Tree Method allows us to determine the asymptotic complexity of recursive functions.
- We can implement and analyze sorting algorithms using the tools we've developed so far

Next Time

- Parallelism & Span
- Trees

Thank you!