Lecture Note (Part 1)

CSCI 6470 Algorithms, Fall 2023

Liming Cai
Department of Computer Science, UGA

September 5, 2023

Part I. Introduction (Chapters 0 and 1)

Topics to be discussed:

- Measuring computational complexities
- Crafting recursive algorithms
- Time complexity of recursive algorithms

1. Measuring complexities

1. Measuring complexities

- what is complexity of algorithms?

1. Measuring complexities

- what is complexity of algorithms? CUP time, memory used

1. Measuring complexities

- what is complexity of algorithms? CUP time, memory used
- why measuring complexity?

1. Measuring complexities

- what is complexity of algorithms? CUP time, memory used
- why measuring complexity? practical usefulness

1. Measuring complexities

- what is complexity of algorithms? CUP time, memory used
- why measuring complexity? practical usefulness
- how to measure?

1. Measuring complexities

- what is complexity of algorithms? CUP time, memory used
- why measuring complexity? practical usefulness
- how to measure? math notions required

1. Measuring complexities

- what is complexity of algorithms? CUP time, memory used
- why measuring complexity? practical usefulness
- how to measure? math notions required
- now what?

1. Measuring complexities

- what is complexity of algorithms? CUP time, memory used
- why measuring complexity? practical usefulness
- how to measure? math notions required
- now what?

  how do we know an algorithm's complexity is acceptable?

1. Measuring complexities

- what is complexity of algorithms? CUP time, memory used
- why measuring complexity? practical usefulness
- how to measure? math notions required
- now what?

  how do we know an algorithm's complexity is acceptable?

  - use absolute standards;

1. Measuring complexities

- what is complexity of algorithms? CUP time, memory used
- why measuring complexity? practical usefulness
- how to measure? math notions required
- now what?

  how do we know an algorithm's complexity is acceptable?

  - use absolute standards;
  - comparison with other algorithms

1. Measuring complexities

1. Measuring complexities

- why are math notions needed?

1. Measuring complexities

- why are math notions needed?

  - different hardwares, system platforms, languages, etc;

1. Measuring complexities

- why are math notions needed?

  - different hardwares, system platforms, languages, etc;

  - amount of data to test issues if based on testing;

1. Measuring complexities

- why are math notions needed?

  - different hardwares, system platforms, languages, etc;

  - amount of data to test issues if based on testing;

  - Math notions would make the issues disappear;

1. Measuring complexities

- why are math notions needed?

  - different hardwares, system platforms, languages, etc;

  - amount of data to test issues if based on testing;

  - Math notions would make the issues disappear;

  how are algorithm/program instructions executed?

1. Measuring complexities

- why are math notions needed?
  - different hardwares, system platforms, languages, etc;
  - amount of data to test issues if based on testing;
  - Math notions would make the issues disappear;

    how are algorithm/program instructions executed?

```
A = B + C;

A = 0;
for (i=1 to N) do
A = A + i;
```

1. Measuring complexities

```
function search (L, x, N);
{
  i = 0;
  while (L[i] != x) AND (i < N)
  {
    i = i + 1;
  }

  if (i < N)
    return (i)
  else
    return (-1)
}
```

1. Measuring complexities

```
function search (L, x, N);
{
  i = 0;
  while (L[i] != x) AND (i < N)
  {
    i = i + 1;
  }

  if (i < N)
    return (i)
  else
    return (-1)
}
```

- to count the number of basic operations;

1. Measuring complexities

```
function search (L, x, N);
{
  i = 0;
  while (L[i] != x) AND (i < N)
  {
    i = i + 1;
  }

  if (i < N)
    return (i)
  else
    return (-1)
}
```

- to count the number of basic operations;
- in terms of input size;

1. Measuring complexities

```
function search (L, x, N);
{
  i = 0;
  while (L[i] != x) AND (i < N)
  {
    i = i + 1;
  }

  if (i < N)
    return (i)
  else
    return (-1)
}
```

- to count the number of basic operations;
- in terms of input size;
- consider the worst cases

1. Measuring complexities

```
function dosomething (N);
{
  x = 0;
  y = 1;
  i = 1;
  while (i < N)
  {
    i = i + 1;
    t = x;
    x = y;
    y = t + x;
  }
  if (N=1)
    return (x);
  else
    return (y);
}
```

1. Measuring complexities

```
function dosomething (N);
{
  x = 0;
  y = 1;
  i = 1;
  while (i < N)
  {
    i = i + 1;
    t = x;
    x = y;
    y = t + x;
  }
  if (N=1)
    return (x);
  else
    return (y);
}
```

- what does this algorithm do?

1. Measuring complexities

```
function dosomething (N);
{
  x = 0;
  y = 1;
  i = 1;
  while (i < N)
  {
    i = i + 1;
    t = x;
    x = y;
    y = t + x;
  }
  if (N=1)
    return (x);
  else
    return (y);
}
```

- what does this algorithm do?
- what is the running time in terms of N?

1. Measuring complexities

```
function dosomething (N);
{
  x = 0;
  y = 1;
  i = 1;
  while (i < N)
  {
    i = i + 1;
    t = x;
    x = y;
    y = t + x;
  }
  if (N=1)
    return (x);
  else
    return (y);
}
```

- what does this algorithm do?
- what is the running time in terms of `N`?

1. Measuring complexities

In-classroom Exercise: Write Insertion Sort Algorithm and analyze its worst case running time as a function of the sorted list length n.

1. Measuring complexities

In-classroom Exercise: Write Insertion Sort Algorithm and analyze its worst case running time as a function of the sorted list length n.

More (take-home) exercises

1. understand the idea of "bubble sort";
2. write your own bubble sort algorithm;
3. analyze the worst case running time of your algorithm;
4. bring an algorithm (not too simple, not too complicated) to the next class.

1. Measuring complexities

(1) Write Insertion Sort algorithm;

1. Measuring complexities

(1) Write Insertion Sort algorithm;

(2) Analyze time complexity;

1. Measuring complexities

(1) Write Insertion Sort algorithm;

(2) Analyze time complexity;

Two notational issues:

1. Measuring complexities

(1) Write Insertion Sort algorithm;

(2) Analyze time complexity;

Two notational issues:

- pseudocode;

1. Measuring complexities

(1) Write Insertion Sort algorithm;

(2) Analyze time complexity;

Two notational issues:

- pseudocode;
- function $T(n)$ is the **worst-case** time;

1. Measuring complexities

(1) Write Insertion Sort algorithm;

(2) Analyze time complexity;

Two notational issues:

- pseudocode;
- function $T(n)$ is the **worst-case** time;

  can $T(n)$ be simplified?

1. Measuring complexities

(1) Write Insertion Sort algorithm;

(2) Analyze time complexity;

Two notational issues:

- pseudocode;
- function $T(n)$ is the **worst-case** time;

  can $T(n)$ be simplified?
  – upper-bounded by a "cleaner function"

1. Measuring complexities

(1) Write Insertion Sort algorithm;

(2) Analyze time complexity;

Two notational issues:

- pseudocode;
- function $T(n)$ is the **worst-case** time;

  can $T(n)$ be simplified?
  – upper-bounded by a "cleaner function"
  – what do the curves of the two function look like?

1. Measuring complexities

Definition of big-O

1. Measuring complexities

Definition of big-O

Let $f(n)$ and $g(n)$ be two functions in n.

1. Measuring complexities

Definition of big-O

Let $f(n)$ and $g(n)$ be two functions in n.
$f(n) = O(g(n))$ if there exist constants c and n_0 such that

1. Measuring complexities

Definition of big-O

Let $f(n)$ and $g(n)$ be two functions in n.
$f(n) = O(g(n))$ if there exist constants c and n_0 such that

$$f(n) \leq cg(n)$$

when $n \geq n_0$.

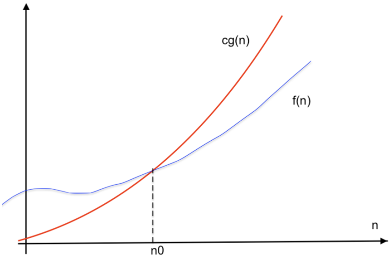
1. Measuring complexities

Definition of big-O

Let $f(n)$ and $g(n)$ be two functions in n.
$f(n) = O(g(n))$ if there exist constants c and n_0 such that

$$f(n) \leq cg(n)$$

when $n \geq n_0$.



1. Measuring complexities

Examples

1. Measuring complexities

Examples

What is the big-O for function $f(n) = 3n^2 - 20n + 100$?

$$\begin{aligned} 3n^2 - 20n + 100 &\leq 3n^2 + 100 \\ &\leq 3n^2 + n^2 \text{ when } n \geq 10 \\ &= 4n^3 \end{aligned}$$

1. Measuring complexities

Examples

What is the big-O for function $f(n) = 3n^2 - 20n + 100$?

$$\begin{aligned} 3n^2 - 20n + 100 &\leq 3n^2 + 100 \\ &\leq 3n^2 + n^2 \text{ when } n \geq 10 \\ &= 4n^3 \end{aligned}$$

We have found $c = 4$ and $n_0 = 10$ such that $f(n) \leq cn^2$.

1. Measuring complexities

Examples

What is the big-O for function $f(n) = 3n^2 - 20n + 100$?

$$\begin{aligned} 3n^2 - 20n + 100 &\le 3n^2 + 100 \\ &\le 3n^2 + n^2 \text{ when } n \ge 10 \\ &= 4n^3 \end{aligned}$$

We have found $c = 4$ and $n_0 = 10$ such that $f(n) \le cn^2$.
I.e., $f(n) = O(n^2)$,

1. Measuring complexities

Examples

What is the big-O for function $f(n) = 3n^2 - 20n + 100$?

$$\begin{aligned} 3n^2 - 20n + 100 &\leq 3n^2 + 100 \\ &\leq 3n^2 + n^2 \text{ when } n \geq 10 \\ &= 4n^3 \end{aligned}$$

We have found $c = 4$ and $n_0 = 10$ such that $f(n) \leq cn^2$.
I.e., $f(n) = O(n^2)$, "$f(n)$ **is big-O of** n^2"

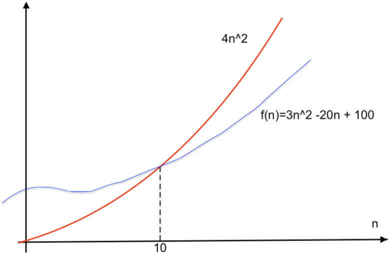
1. Measuring complexities

Examples

What is the big-O for function $f(n) = 3n^2 - 20n + 100$?

$$\begin{aligned} 3n^2 - 20n + 100 &\le 3n^2 + 100 \\ &\le 3n^2 + n^2 \text{ when } n \ge 10 \\ &= 4n^3 \end{aligned}$$

We have found $c = 4$ and $n_0 = 10$ such that $f(n) \le cn^2$.
I.e., $f(n) = O(n^2)$, "$f(n)$ **is big-O of** n^2"



1. Measuring complexities

More examples

1. Measuring complexities

More examples

$$3n^3 + 2n - 6 = O(?)$$

1. Measuring complexities

More examples

$$3n^3 + 2n - 6 = O(?)$$

$$3n \log_2 n + 5n + 7 \log_2 n = O(?)$$

1. Measuring complexities

More examples

$$3n^3 + 2n - 6 = O(?)$$

$$3n \log_2 n + 5n + 7 \log_2 n = O(?)$$

$$2^{2n} + 3 \cdot 2^n = O(?)$$

1. Measuring complexities

More examples

$$3n^3 + 2n - 6 = O(?)$$

$$3n \log_2 n + 5n + 7 \log_2 n = O(?)$$

$$2^{2n} + 3 \cdot 2^n = O(?)$$

$$5 \ln n + 7 \log_{10} n + 2 \log_2 n = O(?)$$

1. Measuring complexities

Notes on big-O

1. Measuring complexities

Notes on big-O

- more than one way to derive the inequality $f(n) \leq cg(n)$;

1. Measuring complexities

Notes on big-O

- more than one way to derive the inequality $f(n) \leq cg(n)$;
- c and n_0 can always be replaced with large numbers;

1. Measuring complexities

Notes on big-O

- more than one way to derive the inequality $f(n) \leq cg(n)$;
- c and n_0 can always be replaced with large numbers;
- for polynomials $f(n)$, $f(n) = O(n^k)$, where k is the largest exponent in positive terms in $f(n)$;

1. Measuring complexities

Notes on big-O

- more than one way to derive the inequality $f(n) \leq cg(n)$;
- c and n_0 can always be replaced with large numbers;
- for polynomials $f(n)$, $f(n) = O(n^k)$, where k is the largest exponent in positive terms in $f(n)$;
- asymptotic, "almost like" $\lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$

1. Measuring complexities

Notes on big-O

- more than one way to derive the inequality $f(n) \leq cg(n)$;
- c and n_0 can always be replaced with large numbers;
- for polynomials $f(n)$, $f(n) = O(n^k)$, where k is the largest exponent in positive terms in $f(n)$;
- asymptotic, "almost like" $\lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$
- it is inherent with algorithms, not with type of algorithms (iterative or recursive)

1. Measuring complexities

In-classroom Exercise: Write Binary Search Algorithm and analyze its worst case running time as a function of the searched list length n.

2. Crafting recursive algorithms

2. Crafting recursive algorithms

- What is recursion?

2. Crafting recursive algorithms

- What is recursion?

  A process to solve a problem in terms of the same process (recursive algorithm)

2. Crafting recursive algorithms

- What is recursion?

  A process to solve a problem in terms of the same process (recursive algorithm)

  A process to define a problem in terms of the same process (recursive definition)

2. Crafting recursive algorithms

- What is recursion?

  A process to solve a problem in terms of the same process (recursive algorithm)

  A process to define a problem in terms of the same process (recursive definition)

- Elements of (meaningful) recursive process

2. Crafting recursive algorithms

- What is recursion?

  A process to solve a problem in terms of the same process (recursive algorithm)

  A process to define a problem in terms of the same process (recursive definition)

- Elements of (meaningful) recursive process

  basic steps;

2. Crafting recursive algorithms

- What is recursion?

  A process to solve a problem in terms of the same process (recursive algorithm)

  A process to define a problem in terms of the same process (recursive definition)

- Elements of (meaningful) recursive process

  basic steps;
  recursive steps;

2. Crafting recursive algorithms

- What is recursion?

  A process to solve a problem in terms of the same process (recursive algorithm)

  A process to define a problem in terms of the same process (recursive definition)

- Elements of (meaningful) recursive process

  basic steps;
  recursive steps;
  changes in problem "size"

2. Crafting recursive algorithms

2. Crafting recursive algorithms

- Why recursive algorithms?

2. Crafting recursive algorithms

- Why recursive algorithms?
- Ideas for recursive algorithms

2. Crafting recursive algorithms

- Why recursive algorithms?
- Ideas for recursive algorithms

  - from well-formulated recursive phenomena; (examples?)

2. Crafting recursive algorithms

- Why recursive algorithms?
- Ideas for recursive algorithms

  - from well-formulated recursive phenomena; (examples?)
  - have to derive a recursive pattern; (examples?)

2. Crafting recursive algorithms

- Why recursive algorithms?
- Ideas for recursive algorithms

  - from well-formulated recursive phenomena; (examples?)
  - have to derive a recursive pattern; (examples?)

    from input data; (examples?)

2. Crafting recursive algorithms

- Why recursive algorithms?
- Ideas for recursive algorithms

  - from well-formulated recursive phenomena; (examples?)
  - have to derive a recursive pattern; (examples?)

    from input data; (examples?)
    from output data (solutions); (examples;)

2. Crafting recursive algorithms

Recursively define a set of objects
(elements in the set are of certain relationships)

2. Crafting recursive algorithms

Recursively define a set of objects
(elements in the set are of certain relationships)

2. Crafting recursive algorithms

Recursively define a set of objects
(elements in the set are of certain relationships)

- the set of natural numbers;

2. Crafting recursive algorithms

Recursively define a set of objects
(elements in the set are of certain relationships)

- the set of natural numbers;
- the set of sums of arithmetic sequences;

2. Crafting recursive algorithms

Recursively define a set of objects
(elements in the set are of certain relationships)

- the set of natural numbers;
- the set of sums of arithmetic sequences;
- the set $\mathcal{T}$ of trees;

  (1) single node $u \in \mathcal{T}$;
  (2) if $t \in \mathcal{T}$, then $t \cup \{(u, v)\} \in \mathcal{T}$, for any $u \in t, v \notin t$;

2. Crafting recursive algorithms

Recursively define a set of objects
(elements in the set are of certain relationships)

2. Crafting recursive algorithms

Recursively define a set of objects
(elements in the set are of certain relationships)

- the set $\mathcal{L}$ of lists;

  (1) $() \in \mathcal{L}$;
  (2) if $l \in \mathcal{L}$, then $l \circ (a) \in \mathcal{L}$, for any element a;

2. Crafting recursive algorithms

Recursively define a set of objects
(elements in the set are of certain relationships)

- the set $\mathcal{L}$ of lists;

  (1) $() \in \mathcal{L}$;
  (2) if $l \in \mathcal{L}$, then $l \circ (a) \in \mathcal{L}$, for any element a;

- the set $\mathcal{SL}$ of sorted lists;

  (1) $() \in \mathcal{SL}$;
  (2) if $l \in \mathcal{SL}$, then $l \circ (a) \in \mathcal{SL}$, for any element $a \geq tail(l)$;

2. Crafting recursive algorithms

Recursively define a set of objects
(elements in the set are of certain relationships)

- the set $\mathcal{L}$ of lists;

  (1) $() \in \mathcal{L}$;
  (2) if $l \in \mathcal{L}$, then $l \circ (a) \in \mathcal{L}$, for any element a;

- the set $\mathcal{SL}$ of sorted lists;

  (1) $() \in \mathcal{SL}$;
  (2) if $l \in \mathcal{SL}$, then $l \circ (a) \in \mathcal{SL}$, for any element $a \geq tail(l)$;

  $tail(l \circ (a)) =_{df} a$.

2. Crafting recursive algorithms

Deriving a recursive pattern (idea)

2. Crafting recursive algorithms

Deriving a recursive pattern (idea)

- Example 1: Linear Search

2. Crafting recursive algorithms

Deriving a recursive pattern (idea)

- Example 1: Linear Search [from recursively defined input!]

2. Crafting recursive algorithms

Deriving a recursive pattern (idea)

- Example 1: Linear Search [from recursively defined input!]

  - the input is a list;

2. Crafting recursive algorithms

Deriving a recursive pattern (idea)

- Example 1: Linear Search [from recursively defined input!]
  - the input is a list;
  - recursively define it?

2. Crafting recursive algorithms

Deriving a recursive pattern (idea)

- Example 1: Linear Search [from recursively defined input!]

  - the input is a list;
  - recursively define it?
  - is a part of the input list also a list? yes.

2. Crafting recursive algorithms

Deriving a recursive pattern (idea)

- Example 1: Linear Search [from recursively defined input!]

  - the input is a list;
  - recursively define it?
  - is a part of the input list also a list? yes.
  - then you get a subproblem to solve;

2. Crafting recursive algorithms

Deriving a recursive pattern (idea)

- Example 1: Linear Search [from recursively defined input!]

  - the input is a list;
  - recursively define it?
  - is a part of the input list also a list? yes.
  - then you get a subproblem to solve;
  - then you get a recursive algorithm!

```
function LinearSearch(L, x, n);
if (n = 0) return ("not found");
else
  if (L[n]=x) return (n);
  else return (LinearSearch(L, x, n-1));
```

2. Crafting recursive algorithms

2. Crafting recursive algorithms

- Example 2. Insertion Sort

2. Crafting recursive algorithms

- Example 2. Insertion Sort [from recursively defined input!]

2. Crafting recursive algorithms

- Example 2. Insertion Sort [from recursively defined input!]

  - input list can recursively defined;

2. Crafting recursive algorithms

- Example 2. Insertion Sort [from recursively defined input!]

  - input list can recursively defined;
  - a part is a sublist, which can sorted first;

2. Crafting recursive algorithms

- Example 2. Insertion Sort [from recursively defined input!]

  - input list can recursively defined;
  - a part is a sublist, which can sorted first;
  - to insert the last element into the sorted sublist;

```
function InsertionSort(L, n);
if (n > 1)
  InsertionSort(L, n-1);
  Insert(L, n) // insert element L[n] into sorted L[1..n-1]

function Insert(L, n)
  if (n > 1)
    if (L[n-1] > L[n])
      swap(L[n-1], L[n]);
      Insert(L, n-1);
```

2. Crafting recursive algorithms

2. Crafting recursive algorithms

- Example 3. Binary Search [from recursively defined input!]

2. Crafting recursive algorithms

- Example 3. Binary Search [from recursively defined input!]

  - think of the whole sorted list as 2 sorted sublists;

2. Crafting recursive algorithms

- Example 3. Binary Search [from recursively defined input!]

  - think of the whole sorted list as 2 sorted sublists;
  - compare the key with the end of the 1st list;

2. Crafting recursive algorithms

- Example 3. Binary Search [from recursively defined input!]

  - think of the whole sorted list as 2 sorted sublists;
  - compare the key with the end of the 1st list;
  - then recursion becomes obvious;

Take-home exercise: use this idea to write a recursive binary search algorithm.

2. Crafting recursive algorithms

2. Crafting recursive algorithms

- Example 4: Selection Sort

2. Crafting recursive algorithms

- Example 4: Selection Sort [from recursively defined output!]

2. Crafting recursive algorithms

- Example 4: Selection Sort [from recursively defined output!]

  - output is a sorted list, defined as
  - concatenation of a sorted list with a largest element

2. Crafting recursive algorithms

- Example 4: Selection Sort [from recursively defined output!]
  - output is a sorted list, defined as
  - concatenation of a sorted list with a largest element
  - a part is a sorted list; how does the unknown algorithm get that part sorted?

2. Crafting recursive algorithms

- Example 4: Selection Sort [from recursively defined output!]
  - output is a sorted list, defined as
  - concatenation of a sorted list with a largest element
  - a part is a sorted list; how does the unknown algorithm get that part sorted? don't care!

2. Crafting recursive algorithms

- Example 4: Selection Sort [from recursively defined output!]
  - output is a sorted list, defined as
  - concatenation of a sorted list with a largest element
  - a part is a sorted list; how does the unknown algorithm get that part sorted? don't care!
  - how does the algorithm get the largest element?

In-classroom exercise: use this idea to write a recursive selection sort algorithm. Note that the step to find the max can also be recursive.

3. Complexity analysis for recursive algorithms

Complexity analysis for iterative algorithms

3. Complexity analysis for recursive algorithms

Complexity analysis for iterative algorithms

- formulate a function $T(n)$ for the algorithm, where n is the "size" of input; e.g., $T(n) = 3n^2 + 25n + 10$;

3. Complexity analysis for recursive algorithms

Complexity analysis for iterative algorithms

- formulate a function $T(n)$ for the algorithm, where n is the "size" of input; e.g., $T(n) = 3n^2 + 25n + 10$;
- show the big-O for $T(n)$, by the definition of big-O, using some basic math knowledge in inequality

3. Complexity analysis for recursive algorithms

Complexity analysis for iterative algorithms

- formulate a function $T(n)$ for the algorithm, where n is the "size" of input; e.g., $T(n) = 3n^2 + 25n + 10$;
- show the big-O for $T(n)$, by the definition of big-O, using some basic math knowledge in inequality

Complexity analysis for recursive algorithms

3. Complexity analysis for recursive algorithms

Complexity analysis for iterative algorithms

- formulate a function $T(n)$ for the algorithm, where n is the "size" of input; e.g., $T(n) = 3n^2 + 25n + 10$;
- show the big-O for $T(n)$, by the definition of big-O, using some basic math knowledge in inequality

Complexity analysis for recursive algorithms

- formulate a recursive function $T(n)$ for the algorithm, e.g., $T(n) = T(n-1) + 5$, (including base cases)

3. Complexity analysis for recursive algorithms

Complexity analysis for iterative algorithms

- formulate a function $T(n)$ for the algorithm, where n is the "size" of input; e.g., $T(n) = 3n^2 + 25n + 10$;
- show the big-O for $T(n)$, by the definition of big-O, using some basic math knowledge in inequality

Complexity analysis for recursive algorithms

- formulate a recursive function $T(n)$ for the algorithm, e.g., $T(n) = T(n-1) + 5$, (including base cases)
- solve $T(n)$ (typically using unfolding or induction) to obtain non-recursive expression, e.g., $T(n) = 2n + 4$;

3. Complexity analysis for recursive algorithms

Complexity analysis for iterative algorithms

- formulate a function $T(n)$ for the algorithm, where n is the "size" of input; e.g., $T(n) = 3n^2 + 25n + 10$;
- show the big-O for $T(n)$, by the definition of big-O, using some basic math knowledge in inequality

Complexity analysis for recursive algorithms

- formulate a recursive function $T(n)$ for the algorithm, e.g., $T(n) = T(n-1) + 5$, (including base cases)
- solve $T(n)$ (typically using unfolding or induction) to obtain non-recursive expression, e.g., $T(n) = 2n + 4$;
- show the big-O for $T(n)$, by the definition of big-O, using some basic math knowledge in inequality

3. Complexity analysis for recursive algorithms

Examples of recursive algorithms:

3. Complexity analysis for recursive algorithms

Examples of recursive algorithms:

- linear search;

3. Complexity analysis for recursive algorithms

Examples of recursive algorithms:

- linear search;
- selection sort;

3. Complexity analysis for recursive algorithms

Examples of recursive algorithms:

- linear search;
- selection sort;
- binary search;

3. Complexity analysis for recursive algorithms

Examples of recursive algorithms:

- linear search;
- selection sort;
- binary search;
- computing the nth Fibonacci number;

3. Complexity analysis for recursive algorithms

```
function LinearSearch(L, x, n);
if (n = 0) return ("not found");
else
  if (L[n]=x) return (n);
  else return (LinearSearch(L, x, n-1));
```

3. Complexity analysis for recursive algorithms

```
function LinearSearch(L, x, n);             <---- T(n)
if (n = 0) return ("not found");            <----- c1
else
  if (L[n]=x) return (n);                   <----- c2
  else return (LinearSearch(L, x, n-1));    <----- c3 + T(n-1)
```

3. Complexity analysis for recursive algorithms

```
function LinearSearch(L, x, n);              <---- T(n)
if (n = 0) return ("not found");             <----- c1
else
  if (L[n]=x) return (n);                    <----- c2
  else return (LinearSearch(L, x, n-1));     <----- c3 + T(n-1)
```

- let $T(n0$ be time function for `LinearSearch(L, x, n);`

3. Complexity analysis for recursive algorithms

```
function LinearSearch(L, x, n);              <---- T(n)
if (n = 0) return ("not found");             <----- c1
else
  if (L[n]=x) return (n);                    <----- c2
  else return (LinearSearch(L, x, n-1));     <----- c3 + T(n-1)
```

- let $T(n0$ be time function for `LinearSearch(L, x, n);`

$$T(n) = c_1$$

3. Complexity analysis for recursive algorithms

```
function LinearSearch(L, x, n);              <---- T(n)
if (n = 0) return ("not found");             <----- c1
else
  if (L[n]=x) return (n);                    <----- c2
  else return (LinearSearch(L, x, n-1));     <----- c3 + T(n-1)
```

- let $T(n0$ be time function for `LinearSearch(L, x, n);`

$$T(n) = c_1 \text{ or } c_2$$

3. Complexity analysis for recursive algorithms

```
function LinearSearch(L, x, n);              <---- T(n)
if (n = 0) return ("not found");             <----- c1
else
  if (L[n]=x) return (n);                    <----- c2
  else return (LinearSearch(L, x, n-1));     <----- c3 + T(n-1)
```

- let $T(n0$ be time function for `LinearSearch(L, x, n);`

$$T(n) = c_1 \text{ or } c_2 \text{ or } c_3 + T(n-1)$$

3. Complexity analysis for recursive algorithms

```
function LinearSearch(L, x, n);                <---- T(n)
if (n = 0) return ("not found");               <----- c1
else
  if (L[n]=x) return (n);                      <----- c2
  else return (LinearSearch(L, x, n-1));       <----- c3 + T(n-1)
```

- let $T(n0$ be time function for `LinearSearch(L, x, n);`

$$T(n) = c_1 \text{ or } c_2 \text{ or } c_3 + T(n-1)$$

$$T(n) = c_3 + T(n-1)$$

3. Complexity analysis for recursive algorithms

```
function LinearSearch(L, x, n);            <---- T(n)
if (n = 0) return ("not found");           <----- c1
else
  if (L[n]=x) return (n);                  <----- c2
  else return (LinearSearch(L, x, n-1));   <----- c3 + T(n-1)
```

- let $T(n0$ be time function for `LinearSearch(L, x, n);`

$$T(n) = c_1 \text{ or } c_2 \text{ or } c_3 + T(n-1)$$

$$T(n) = c_3 + T(n-1)$$

$$T(0) = c_1$$

3. Complexity analysis for recursive algorithms

In-classroom Exercise: Analyzing worst case time complexity for recursive Selection Sort algorithm.

3. Complexity analysis for recursive algorithms

In-classroom Exercise: Analyzing worst case time complexity for recursive Selection Sort algorithm.

```
function SelectionSort(L, n);
 if n > 1
   FindMax(L, n); //find and move the max to the rightmost;
   SelectionSort(L, n-1);

function FindMax(L, n-1);
 if n>1
   FindMax(L, n-1);
   if L[n-1] > L[n]
     swap(L[n-1], L[n]);
```

3. Complexity analysis for recursive algorithms

Let $T(n)$ be time complexity for `SelctionSort(L, n)`;
Let $S(n)$ be time complexity for `FindMax(L, n)`;

3. Complexity analysis for recursive algorithms

Let $T(n)$ be time complexity for `SelctionSort(L, n)`;
Let $S(n)$ be time complexity for `FindMax(L, n)`;

Step 1. formulating complexity functions:

$$T(n) = \begin{cases} S(n) + T(n-1) + a & n > 1 \\ b & n = 1 \end{cases}$$

$$S(n) = \begin{cases} S(n-1) + c & n > 1 \\ d & n = 1 \end{cases}$$

3. Complexity analysis for recursive algorithms

Step 2. Solving for $S(n)$ and $T(n)$ using simple "unfolding" method;

$S(n) = xn + y$ for constants x, y

$S(n) = un^2 + vn + w$ for constants u, v, w;

3. Complexity analysis for recursive algorithms

Step 2. Solving for $S(n)$ and $T(n)$ using simple "unfolding" method;

$S(n) = xn + y$ for constants x, y

$S(n) = un^2 + vn + w$ for constants u, v, w;

Step 3. Express $T(n)$ in terms of big-O:

$$T(n) = O(n^2)$$

3. Complexity analysis for recursive algorithms

In-classroom Exercise: Analyzing worst case time complexity for recursive binary search algorithm.

3. Complexity analysis for recursive algorithms

In-classroom Exercise: Analyzing worst case time complexity for recursive Fibonacci algorithm.

```
function Fib(N);                         <-- T(n)
  if (n = 1)
    return (0)                           <-- c1
  else
    if (n = 2)
      return (1)                         <-- c2
    else
      return (Fib(n-1) + Fib(n-2))       <-- T(n-1) + T(n-2) + a
```

$$T(n) = \begin{cases} T(n-1) + T(n-2) + a & n > 2 \\ b & n = 0 \text{ or } n = 1 \end{cases}$$

3. Complexity analysis for recursive algorithms

Issues in assessing the time complexity;

3. Complexity analysis for recursive algorithms

Issues in assessing the time complexity;

- sometime (not necessarily precise) estimation is enough;

3. Complexity analysis for recursive algorithms

Issues in assessing the time complexity;

- sometime (not necessarily precise) estimation is enough;
- to see how bad an algorithm is, lower bound (what is it?), instead of upper bound, is important;

3. Complexity analysis for recursive algorithms

Issues in assessing the time complexity;

- sometime (not necessarily precise) estimation is enough;
- to see how bad an algorithm is, lower bound (what is it?), instead of upper bound, is important;
- `Fib` has $T(N)$ exponential time due to duplicate computations;

3. Complexity analysis for recursive algorithms

Issues in assessing the time complexity;

- sometime (not necessarily precise) estimation is enough;
- to see how bad an algorithm is, lower bound (what is it?), instead of upper bound, is important;
- `Fib` has $T(N)$ exponential time due to duplicate computations;
- are there recursive algorithms for the Fibonacci problem?

3. Complexity analysis for recursive algorithms

Issues in assessing the time complexity;

- sometime (not necessarily precise) estimation is enough;
- to see how bad an algorithm is, lower bound (what is it?), instead of upper bound, is important;
- `Fib` has $T(N)$ exponential time due to duplicate computations;
- are there recursive algorithms for the Fibonacci problem?
- $T(N)$ for `Fib`, where N is value;
  what will happen if we measure $T(n)$, where $n = |N|$?

3. Complexity analysis for recursive algorithms

Issues in assessing the time complexity;

- sometime (not necessarily precise) estimation is enough;
- to see how bad an algorithm is, lower bound (what is it?), instead of upper bound, is important;
- `Fib` has $T(N)$ exponential time due to duplicate computations;
- are there recursive algorithms for the Fibonacci problem?
- $T(N)$ for `Fib`, where N is value;
  what will happen if we measure $T(n)$, where $n = |N|$?
- operation "$+$" is assumed taking constant time? does it?

3. Complexity analysis for recursive algorithms

In-classroom Exercise: Design a recursive algorithm for the Fibonacci problem.

3. Complexity analysis for recursive algorithms

In-classroom Exercise: Design a recursive algorithm for the Fibonacci problem. *Hint: instead of returning the Nth Fibonacci number, return the list of first N Fibonacci numbers.*

4. More about big-O

Categories of big-O functions

4. More about big-O

Categories of big-O functions

- logarithmic: $O(\log_2 n)$; $O(\log_2 n)^4)$; $O(\log_2(\log_2 n))$;

4. More about big-O

Categories of big-O functions

- logarithmic: $O(\log_2 n)$; $O(\log_2 n)^4)$; $O(\log_2(\log_2 n))$;
- polynomial: $O(n)$, $O(\sqrt{n})$; $O(n \log_2 n)$; $O(n^c)$, $c > 0$;

4. More about big-O

Categories of big-O functions

- logarithmic: $O(\log_2 n)$; $O(\log_2 n)^4)$; $O(\log_2(\log_2 n))$;
- polynomial: $O(n)$, $O(\sqrt{n})$; $O(n \log_2 n)$; $O(n^c)$, $c > 0$;
- sub-exponential: $O(2^{(\log_2 n)^k})$, $k > 1$; $O(n^{(\log_2 n)^k})$, $k \geq 1$;

4. More about big-O

Categories of big-O functions

- logarithmic: $O(\log_2 n)$; $O(\log_2 n)^4)$; $O(\log_2(\log_2 n))$;
- polynomial: $O(n)$, $O(\sqrt{n})$; $O(n \log_2 n)$; $O(n^c)$, $c > 0$;
- sub-exponential: $O(2^{(\log_2 n)^k})$, $k > 1$; $O(n^{(\log_2 n)^k})$, $k \geq 1$;
- exponential: $O(2^n)$; $O(2^{cn})$, $c > 1$; $O(2^{n^c})$; $c > 1$; $O(2^{2^n})$;

4. More about big-O

Composite rules for big-O

4. More about big-O

Composite rules for big-O

- $c \times O(f(n)) = O(f(n))$;

4. More about big-O

Composite rules for big-O

- $c \times O(f(n)) = O(f(n))$;
- $O(f(n)) + O(g(n)) =?$;

4. More about big-O

Composite rules for big-O

- $c \times O(f(n)) = O(f(n))$;
- $O(f(n)) + O(g(n)) =?$;
- $O(f(n)) \times O(g(n)) =?$;

4. More about big-O

Composite rules for big-O

- $c \times O(f(n)) = O(f(n))$;
- $O(f(n)) + O(g(n)) =?$;
- $O(f(n)) \times O(g(n)) =?$;
- $(O(f(n)))^c = O(f(n)^c)$;

4. Proof by math induction

4. Proof by math induction

- remember recursive definition of set $\mathcal{N} = \{1, 2, \dots\}$?

4. Proof by math induction

- remember recursive definition of set $\mathcal{N} = \{1, 2, \dots\}$?

  creates a "bucket" $\mathcal{N}$ to include one number at a time;
  $k + 1$ is placed in $\mathcal{N}$ if
  k is already in $\mathcal{N}$;

4. Proof by math induction

- remember recursive definition of set $\mathcal{N} = \{1, 2, \dots\}$?

  creates a "bucket" $\mathcal{N}$ to include one number at a time;
  $k + 1$ is placed in $\mathcal{N}$ if
  k is already in $\mathcal{N}$;

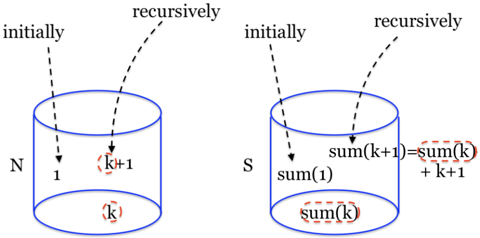
- recursive definition of set $\mathcal{S}$ of $sum(k)$'s, $\forall k \in \mathcal{N}$;
  $sum(k + 1) = sum(k) + k$ is placed in $\mathcal{S}$ if
  $sum(k)$ is already in $\mathcal{S}$;

4. Proof by math induction

- remember recursive definition of set $\mathcal{N} = \{1, 2, \dots\}$?

  creates a "bucket" $\mathcal{N}$ to include one number at a time;
  $k + 1$ is placed in $\mathcal{N}$ if
  k is already in $\mathcal{N}$;

- recursive definition of set $\mathcal{S}$ of $sum(k)$'s, $\forall k \in \mathcal{N}$;
  $sum(k+1) = sum(k) + k$ is placed in $\mathcal{S}$ if
  $sum(k)$ is already in $\mathcal{S}$;



4. Proof by math induction

4. Proof by math induction

- Now suppose for any item $sum(n)$, before it is put in $\mathcal{S}$, it needs to be verified

$$sum(n) = \frac{n}{2}(n+1)$$

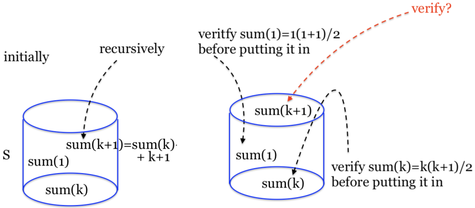
what would you do?

4. Proof by math induction

- Now suppose for any item $sum(n)$, before it is put in $\mathcal{S}$, it needs to be verified

$$sum(n) = \frac{n}{2}(n+1)$$

what would you do?

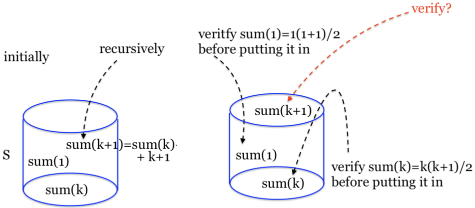


4. Proof by math induction

- Now suppose for any item $sum(n)$, before it is put in $\mathcal{S}$, it needs to be verified

$$sum(n) = \frac{n}{2}(n+1)$$

what would you do?



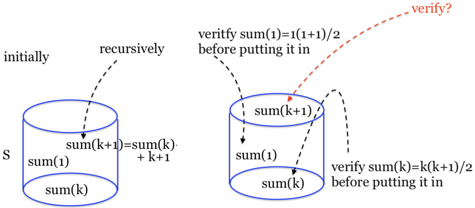
- if $sum(k) = \frac{k}{2}(k+1)$ has been verified,

4. Proof by math induction

- Now suppose for any item $sum(n)$, before it is put in $\mathcal{S}$, it needs to be verified

$$sum(n) = \frac{n}{2}(n+1)$$

what would you do?



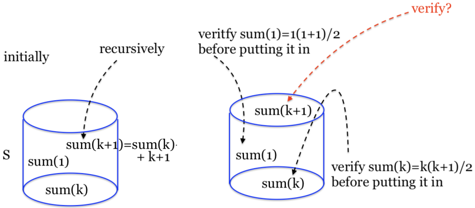
- if $sum(k) = \frac{k}{2}(k+1)$ has been verified, verifying $sum(k+1) = \frac{k+1}{2}(k+1+1)$ can be done (conveniently) by using recursive formula $sum(k+1) = sum(k) + k + 1$.

4. Proof by math induction

- Now suppose for any item $sum(n)$, before it is put in $\mathcal{S}$, it needs to be verified

$$sum(n) = \frac{n}{2}(n+1)$$

what would you do?



- if $sum(k) = \frac{k}{2}(k+1)$ has been verified, verifying $sum(k+1) = \frac{k+1}{2}(k+1+1)$ can be done (conveniently) by using recursive formula $sum(k+1) = sum(k) + k + 1$. This is induction!

4. Proof by math induction

Remember this page!!

4. Proof by math induction

Remember this page!!

To prove objects related to n, e.g., $sum(n)$, to have some property $\mathcal{P}(n)$,

4. Proof by math induction

Remember this page!!

To prove objects related to n, e.g., $sum(n)$, to have some property $\mathcal{P}(n)$, e.g., $sum(n) = \frac{n}{2}(n+1)$ for all $n \geq b$;

4. Proof by math induction

Remember this page!!

To prove objects related to n, e.g., $sum(n)$, to have some property $\mathcal{P}(n)$, e.g., $sum(n) = \frac{n}{2}(n+1)$ for all $n \geq b$;

- need to identify a recursive relation in the objects,

4. Proof by math induction

Remember this page!!

To prove objects related to n, e.g., $sum(n)$, to have some property $\mathcal{P}(n)$, e.g., $sum(n) = \frac{n}{2}(n+1)$ for all $n \geq b$;

- need to identify a recursive relation in the objects, e.g., $sum(k+1) = sum(k) + k + 1$;

4. Proof by math induction

Remember this page!!

To prove objects related to n, e.g., $sum(n)$, to have some property $\mathcal{P}(n)$, e.g., $sum(n) = \frac{n}{2}(n+1)$ for all $n \geq b$;

- need to identify a recursive relation in the objects, e.g., $sum(k+1) = sum(k) + k + 1$;
- need to prove property $\mathcal{P}(b)$ holds for some initial case b,

4. Proof by math induction

Remember this page!!

To prove objects related to n, e.g., $sum(n)$, to have some property $\mathcal{P}(n)$, e.g., $sum(n) = \frac{n}{2}(n+1)$ for all $n \geq b$;

- need to identify a recursive relation in the objects, e.g., $sum(k+1) = sum(k) + k + 1$;
- need to prove property $\mathcal{P}(b)$ holds for some initial case b, e.g., for $n = 1$, $sum(1) = \frac{1}{2}(1+1)$

4. Proof by math induction

Remember this page!!

To prove objects related to n, e.g., $sum(n)$, to have some property $\mathcal{P}(n)$, e.g., $sum(n) = \frac{n}{2}(n+1)$ for all $n \geq b$;

- need to identify a recursive relation in the objects, e.g., $sum(k+1) = sum(k) + k + 1$;
- need to prove property $\mathcal{P}(b)$ holds for some initial case b, e.g., for $n = 1$, $sum(1) = \frac{1}{2}(1+1)$
- need to prove implication $\mathcal{P}(k) \to \mathcal{P}(k+1)$;

4. Proof by math induction

Remember this page!!

To prove objects related to n, e.g., $sum(n)$, to have some property $\mathcal{P}(n)$, e.g., $sum(n) = \frac{n}{2}(n+1)$ for all $n \geq b$;

- need to identify a recursive relation in the objects, e.g., $sum(k+1) = sum(k) + k + 1$;
- need to prove property $\mathcal{P}(b)$ holds for some initial case b, e.g., for $n = 1$, $sum(1) = \frac{1}{2}(1+1)$
- need to prove implication $\mathcal{P}(k) \to \mathcal{P}(k+1)$; e.g., $sum(k) = \frac{k}{2}(k+1) \to sum(k+1) = \frac{k+1}{2}(k+1+1)$

4. Proof by math induction

Remember this page!!

To prove objects related to n, e.g., $sum(n)$, to have some property $\mathcal{P}(n)$, e.g., $sum(n) = \frac{n}{2}(n+1)$ for all $n \geq b$;

- need to identify a recursive relation in the objects, e.g., $sum(k+1) = sum(k) + k + 1$;
- need to prove property $\mathcal{P}(b)$ holds for some initial case b, e.g., for $n = 1$, $sum(1) = \frac{1}{2}(1+1)$
- need to prove implication $\mathcal{P}(k) \rightarrow \mathcal{P}(k+1)$; e.g., $sum(k) = \frac{k}{2}(k+1) \rightarrow sum(k+1) = \frac{k+1}{2}(k+1+1)$

  i.e., assume $sum(k) = \frac{k}{2}(k+1)$ holds,

4. Proof by math induction

Remember this page!!

To prove objects related to n, e.g., $sum(n)$, to have some property $\mathcal{P}(n)$, e.g., $sum(n) = \frac{n}{2}(n+1)$ for all $n \geq b$;

- need to identify a recursive relation in the objects, e.g., $sum(k+1) = sum(k) + k + 1$;
- need to prove property $\mathcal{P}(b)$ holds for some initial case b, e.g., for $n = 1$, $sum(1) = \frac{1}{2}(1+1)$
- need to prove implication $\mathcal{P}(k) \rightarrow \mathcal{P}(k+1)$; e.g., $sum(k) = \frac{k}{2}(k+1) \rightarrow sum(k+1) = \frac{k+1}{2}(k+1+1)$

  i.e., assume $sum(k) = \frac{k}{2}(k+1)$ holds,
  prove $sum(k+1) = \frac{k+1}{2}(k+1+1)$ holds

4. Proof by math induction

In-classroom Exercise: Prove that the for all $n \geq 0$, $\alpha \neq 1$

$$1 + \alpha + \alpha^2 + \cdots + \alpha^n = \frac{1 - \alpha^{n+1}}{1 - \alpha}$$

4. Proof by math induction

In-classroom Exercise: Prove that the for all $n \geq 0$, $\alpha \neq 1$

$$1 + \alpha + \alpha^2 + \cdots + \alpha^n = \frac{1 - \alpha^{n+1}}{1 - \alpha}$$

- what is the recursive relation here?

4. Proof by math induction

In-classroom Exercise: Prove that the for all $n \geq 0$, $\alpha \neq 1$

$$1 + \alpha + \alpha^2 + \cdots + \alpha^n = \frac{1 - \alpha^{n+1}}{1 - \alpha}$$

- what is the recursive relation here?
- base case proof?

4. Proof by math induction

In-classroom Exercise: Prove that the for all $n \geq 0$, $\alpha \neq 1$

$$1 + \alpha + \alpha^2 + \cdots + \alpha^n = \frac{1 - \alpha^{n+1}}{1 - \alpha}$$

- what is the recursive relation here?
- base case proof?
- implication proof? assumption $\rightarrow$ induction

4. Proof by math induction

A variant of math induction

4. Proof by math induction

A variant of math induction

To prove property $\mathcal{P}(n)$ holds for all $n \geq b$; it suffices

4. Proof by math induction

A variant of math induction

To prove property $\mathcal{P}(n)$ holds for all $n \geq b$; it suffices

- to prove $\mathcal{P}(b)$, for some initial case b; and

4. Proof by math induction

A variant of math induction

To prove property $\mathcal{P}(n)$ holds for all $n \geq b$; it suffices

- to prove $\mathcal{P}(b)$, for some initial case b; and
- to assume $\mathcal{P}(n)$ holds for all $n = b, b+1, \ldots, k$, and

4. Proof by math induction

A variant of math induction

To prove property $\mathcal{P}(n)$ holds for all $n \geq b$; it suffices

- to prove $\mathcal{P}(b)$, for some initial case b; and
- to assume $\mathcal{P}(n)$ holds for all $n = b, b+1, \ldots, k$, and
- prove $\mathcal{P}(k+1)$ holds;

4. Proof by math induction

In-classroom Exercise: Prove that the nth Fibonacci number

$$F_n \leq 1.8^n$$

4. Proof by math induction

Apply math induction to prove $T(n) = O(\ )$, when $T(n)$ is recursively defined!

4. Proof by math induction

Apply math induction to prove $T(n) = O(\ )$, when $T(n)$ is recursively defined!

- Step 1: guess a function $g(n)$ for $T(n) = O(g(n))$;

4. Proof by math induction

Apply math induction to prove $T(n) = O(\ )$, when $T(n)$ is recursively defined!

- Step 1: guess a function $g(n)$ for $T(n) = O(g(n))$;
- Step 2: there exist constants $c > 0, n_0 > 0$ such that

$$T(n) \leq cg(n) \qquad \text{when } n \geq n_0$$

4. Proof by math induction

In-classroom Exercise: What is big-O for $T(n)$, the time function of `LinearSearch`? where $T(n)$ was derived as

$$T(n) = \begin{cases} T(n-1) + a & n > 0 \\ b & n = 0 \end{cases}$$

4. Proof by math induction