Algorithm Efficiency

Computer Science 111
Boston University

Vahid Azadeh Ranjbar, Ph.D.

Algorithm Efficiency

- This semester, we've developed algorithms for many tasks.
- For a given task, there may be more than one algorithm that works.
- When choosing among algorithms, one important factor is their relative *efficiency*.
  - space efficiency: how much memory an algorithm requires
  - time efficiency: how quickly an algorithm executes
    - how many "operations" it performs

Two Approaches to the Same Problem

```
def power(b, p):
    """ returns b raised to the p power
        inputs: b is a number (int or float)
                p is a non-negative integer
    """
```

- Here's how we recursively reduced b^p earlier in the semester:

  $b^p = b * b^{p-1}$

  - for example:

    $2^{10} = 2 * 2^9$

    $2^9 = 2 * 2^8$

    ...

- Base case: $b^0 = 1$

Recursively Raising a Number to a Power

```
def power(b, p):
    """ returns b raised to the p power
        inputs: b is a number (int or float)
                p is a non-negative integer
    """
    if p == 0:           # base case
        return 1

    else:
        pow_rest = power(b, p-1)
        return b * pow_rest
```

Two Approaches to the Same Problem (cont.)

- Each recursive call only reduces the exponent by 1.
- How many times will `power()` be called when computing 2^{1000}?
  1001

Two Approaches to the Same Problem (cont.)

- There's another way to reduce this problem.
- When the exponent is ***even***, we can do this:
  $b^p = (b^{p/2}) * (b^{p/2})$
  - for example:
    $2^{10} = 2^5 * 2^5$
- When the exponent is ***odd***, we can do this:
  $b^p = b * (b^{p/2}) * (b^{p/2})$ (using integer division: `p//2`)
  - for example:
    $2^5 = 2 * 2^2 * 2^2$
- Each recursive call cuts the exponent in half!

A More Efficient Power!

$b^p = (b^{p/2}) * (b^{p/2})$ when p is even and greater than 0

$b^p = b * (b^{p/2}) * (b^{p/2})$ when p is odd and greater than 0

```
def power2(b, p):
    """ docstring goes here...   """
    if p == 0:                  # base case
        return 1
    else:                       # recursive case
        pow_rest = power2(b, p // 2)
        if p % 2 == 0:
            return pow_rest * pow_rest
        else:
            return b * pow_rest * pow_rest
```

A More Efficient Power! (cont.)

- How many times will `power2()` be called when computing 2^{1000}?
  11

```
power2(2, 1000)
  power2(2, 500)
    power2(2, 250)
      power2(2, 125)
        power2(2, 62)
          power2(2, 31)
            power2(2, 15)
              power2(2, 7)
                power2(2, 3)
                  power2(2, 1)
                    power2(2, 0)
```

- Much more efficient than the original `power()` when the starting exponent is large!

An Inefficient Version of power2

- What's wrong with the following version of `power2()`?

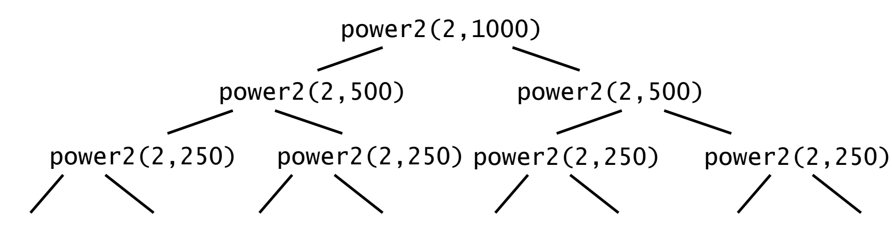
```
def power2bad(b, p):
  """ docstring goes here...     """
  if p == 0:                  # base case
      return 1
  else:                       # recursive case
      if p % 2 == 0:
          return power2(b, p//2) * power2(b, p//2)
      else:
          return b * power2(b, p//2) * power2(b, p//2)
```

```
pow_rest = power2(b, p // 2)
      if p % 2 == 0:
            return pow_rest * pow_rest
        else:
            return b * pow_rest * pow_rest
```


An Inefficient Version of power2

- What's wrong with the following version of `power2()`?

```
def power2bad(b, p):
  """ docstring goes here...     """
  if p == 0:                  # base case
      return 1
  else:                       # recursive case
      if p % 2 == 0:
          return power2(b, p//2) * power2(b, p//2)
      else:
          return b * power2(b, p//2) * power2(b, p//2)
```

Example of Comparing Algorithms

- Consider the problem of finding a phone number in a phonebook.
- Let's informally compare the time efficiency of two algorithms for this problem.

Algorithm 1 for Finding a Phone Number

```
def find_number1(person, phonebook):
    for p in range(1, phonebook.num_pages + 1):
        if person is found on page p:
            return the person's phone number

    return None
```

- If there were 1,000 pages in the phonebook, how many pages would this look at in the worst case? 1,000
- What if there were 1,000,000 pages? 1,000,000
- The running time of this algorithm "grows proportionally" to *n* (*n* = # of pages).

Algorithm 2 for Finding a Phone Number

```
def find_number2(person, phonebook) {
    min = 1
    max = phonebook.num_pages
    while min <= max:
        mid = (min + max) // 2     # the middle page
        if person is found on page mid:
            return the person's number
        elif person comes earlier in phonebook:
            max = mid - 1
        else:
            min = mid + 1
    return None
```

- If there were 1,000 pages in the phonebook, how many pages would this look at in the worst case? approx. 10
- What if there were 1,000,000 pages? approx. 20
- The running time "grows proportionally" to $\log_2 n$ (n = # of pages).

Searching a Collection of Data

- The phonebook problem is one example of a common task: searching for an item in a collection of data.
  - another example: searching for a value in a list
- Algorithm 1 is known as *sequential search*.
- Algorithm 2 is known as *binary search*.
  - only works if the items in the data collection are sorted

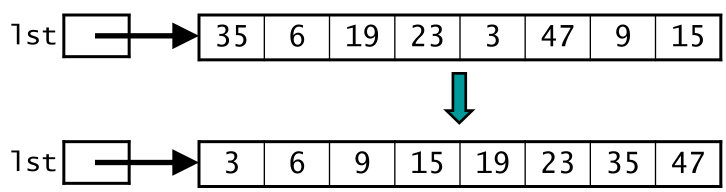
Searching a Collection of Data

- The phonebook problem is one example of a common task: searching for an item in a collection of data.
  - another example: searching for a value in a list
- Algorithm 1 is known as *sequential search*.
- Algorithm 2 is known as *binary search*.
- For large collections of data, binary search is significantly faster than sequential search.

Algorithm 2 works only if the items in the data collection are sorted.

Sorting a Collection of Data

- It's often useful to be able to sort the items in a list.
- Example:



- Many algorithms have been developed for this purpose.
  - CS 112 looks at a number of them
- For large collections of data, some sorting algorithms are *much* faster than others.
  - we can see this by comparing two of them

Selection Sort

- Basic idea:
  - consider the positions in the list from left to right
  - for each position, find the element that belongs there and swap it with the element that's currently there
- Example:

0	1	2	3	4
15	6	2	12	4

Selection Sort

- Basic idea:
  - consider the positions in the list from left to right
  - for each position, find the element that belongs there and swap it with the element that's currently there
- Example:

0	1	2	3	4
15	6	*2*	12	4

0	1	2	3	4
2	6	15	12	4

Selection Sort

- Basic idea:
  - consider the positions in the list from left to right
  - for each position, find the element that belongs there and swap it with the element that's currently there
- Example:

0	1	2	3	4
15	6	2	12	4

0	1	2	3	4
2	6	15	12	4

0	1	2	3	4
2	4	15	12	6

Selection Sort

- Basic idea:
  - consider the positions in the list from left to right
  - for each position, find the element that belongs there and swap it with the element that's currently there
- Example:

0	1	2	3	4
15	6	2	12	4

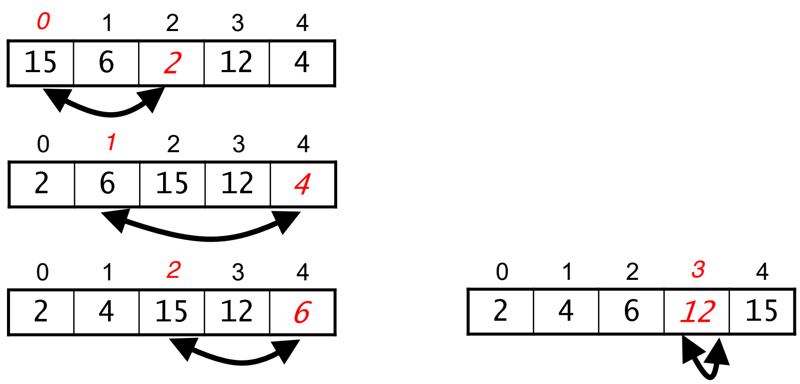
0	1	2	3	4
2	6	15	12	4

0	1	2	3	4
2	4	15	12	6

0	1	2	3	4
2	4	6	12	15

Selection Sort

- Basic idea:
  - consider the positions in the list from left to right
  - for each position, find the element that belongs there and swap it with the element that's currently there
- Example:



Why don't we need to consider position 4?

If we're using selection sort to sort `[24, 8, 5, 2, 17, 10, 7]` what will the list look like after we select elements for the <u>first three positions</u>?

A. `[2, 5, 7, 24, 17, 10, 8]`

B. `[2, 5, 7, 8, 24, 17, 10]`

C. `[5, 8, 24, 2, 17, 10, 7]`

D. `[2, 5, 8, 24, 17, 10, 7]`

E. none of these

Basic idea:
- consider the positions in the list from left to right
- for each position:
  - find the element that belongs there
  - swap it with the element that's currently there

If we're using selection sort to sort
`[24, 8, 5, 2, 17, 10, 7]`
what will the list look like after we select
elements for the <u>first three positions</u>?

A. `[2, 5, 7, 24, 17, 10, 8]`

B. `[2, 5, 7, 8, 24, 17, 10]`

C. `[5, 8, 24, 2, 17, 10, 7]`

D. `[2, 5, 8, 24, 17, 10, 7]`

E. none of these

```
  0   1   2   3   4   5   6
[24,  8,  5,  2, 17, 10,  7]

  0   1   2   3   4   5   6
[ 2,  8,  5, 24, 17, 10,  7]
```

If we're using selection sort to sort
`[24, 8, 5, 2, 17, 10, 7]`
what will the list look like after we select
elements for the <u>first three positions</u>?

A. `[2, 5, 7, 24, 17, 10, 8]`

B. `[2, 5, 7, 8, 24, 17, 10]`

C. `[5, 8, 24, 2, 17, 10, 7]`

D. `[2, 5, 8, 24, 17, 10, 7]`

E. none of these

```
  0   1   2   3   4   5   6
[24,  8,  5,  2, 17, 10,  7]

  0   1   2   3   4   5   6
[ 2,  8,  5, 24, 17, 10,  7]

  0   1   2   3   4   5   6
[ 2,  5,  8, 24, 17, 10,  7]
```

If we're using selection sort to sort
`[24, 8, 5, 2, 17, 10, 7]`
what will the list look like after we select elements for the <u>first three positions</u>?

A. `[2, 5, 7, 24, 17, 10, 8]`

B. `[2, 5, 7, 8, 24, 17, 10]`

C. `[5, 8, 24, 2, 17, 10, 7]`

D. `[2, 5, 8, 24, 17, 10, 7]`

E. none of these

```
  0   1   2   3   4   5   6
[24,  8,  5,  2, 17, 10,  7]

  0   1   2   3   4   5   6
[ 2,  8,  5, 24, 17, 10,  7]

  0   1   2   3   4   5   6
[ 2,  5,  8, 24, 17, 10,  7]

  0   1   2   3   4   5   6
[ 2,  5,  7, 24, 17, 10,  8]
```

Quicksort

- Another possible sorting algorithm is called quicksort.
- It uses recursion to "divide-and conquer":
  - *divide:* rearrange the elements so that we end up with two sublists that meet the following criterion:
    - *each element in the left list <= each element in the right list*

    example:

12	8	14	4	6	13

    →

6	8	4	14	12	13

  - *conquer:* apply quicksort recursively to the sublists, stopping when a sublist has a single element
  - note: when the recursive calls return, nothing else needs to be done to "combine" the two sublists!

Comparing Selection Sort and Quicksort

- Selection sort's running time "grows proportionally to" n^2,
(n = length of list).
  - make the list 2x longer → the running time will be ~4x longer
  - make the list 3x longer → the running time will be ~9x longer
  - make the list 4x longer → the running time will be ~16x longer
- Quicksort's running time "grows proportionally to" $n \log_2 n$.
  - we've seen that $\log_2 n$ grows much more slowly than n
  - thus, $n \log_2 n$ grows much more slowly than n^2
- For large lists, quicksort is significantly faster than selection sort.