北京航空航天大學
BEIHANG UNIVERSITY

Introduction to Computer Science and Programming

Lecture 8

Sebastian Wandelt (小塞)

Beihang University

Outline

- Recap
- Complexity Analysis / Solving Recurrences
  - Integer Multiplication
  - Median and Selection

Recap

Recap

- What do you remember about last week's lecture?

Recap sorting

- What was the best/worst time complexity of Bubble sort, Insert sort and Merge sort?

Recap sorting

- What does this code do?

```
def f(L):
    if len(L)==1:
        return L
    else:
        part=f(L[:-1])
        i=len(part)
        while i>0 and (L[-1]<part[i-1]):
            i=i-1
        return part[0:i]+[L[-1]]+part[i:]

L=[6,2,8,1,9,5]
print(f(L))
```

Recap sorting

- What does this code do?

Recursive insertion sort
Time complexity?


```
def f(L):
    if len(L)==1:
        return L
    else:
        part=f(L[:-1])
        i=len(part)
        while i>0 and (L[-1]<part[i-1]):
            i=i-1
        return part[0:i]+[L[-1]]+part[i:]

L=[6,2,8,1,9,5]
print(f(L))
```

Sorting as dancing

- https://www.i-programmer.info/programming/theory/3531-sorting-algorithms-as-dances.html

Big O - Decisions

- Examples:
  - n=O(n*n)?
  - log(n)=O(sqrt(n))

Transforming Big O to limits and using L'Hopital

Let f(n) and g(n) be two real function then, $f(n) = O(g(n))$ is equivalent to

$$\exists\, c \in \mathbb{R} : \lim_{n \to \infty} \frac{f(n)}{g(n)} = c$$

where, $c \geqslant 0$

(*Here c is finite or* ***can be zero***)

So, if you prove above *lim* is equal to 0 then you can say that $f(n) = O(g(n))$. But, if it's not equal to 0 and $c > 0$, then you can't say for sure that $f(n)$ is not $O(g(n))$

Theorem

(l'Hôpital's Rule) Let f and g, if the limit between the quotient $\frac{f(n)}{g(n)}$ exists, it is equal to the limit of the derivative of the denominator and the numerator.

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \lim_{n \to \infty} \frac{f'(n)}{g'(n)}$$

Big O

- Examples:
  - n=O(n*n)?
  - log(n)=O(sqrt(n))

By l'Hopital's Rule,

$$\lim_{x \to \infty} \frac{\ln x}{\sqrt{x}} = \lim_{x \to \infty} \frac{\frac{1}{x}}{\frac{1}{2\sqrt{x}}} = \lim_{x \to \infty} \frac{2\sqrt{x}}{x} = \lim_{x \to \infty} \frac{2}{\sqrt{x}} = 0$$

Big O

- Can you think of one f and one g, such that
  f!=O(g) and g!=O(f)?

Integer Multiplication

Why do we look at a new problem?

- Not everything in life is about sorting, is it?
- We will come back to sorting soon enough, don't worry ☺

Integer Multiplication

1 x 2 = 2

13 x 24 = 312

1357 x 2468 = 3,349,076

13579246801593726048 x 24680135792604815937 = ???

n

How long would it take you to solve this problem?

Integer Multiplication

1 x 2 = 2

13 x 24 = 312

1357 x 2468 = 3,349,076

13579246801593726048 x 24680135792604815937 = ???

n

How long would it take you to solve this problem?

About n^2 one-digit operations.

- At most n^2 multiplications
- At most n^2 additions (for carries)
- Addition of n different 2n-digit numbers

Key remark for the next slides

- Addition of two numbers is much simpler than multiplication!
- We assume, addition is, essentially, **for free**

Integer Multiplication

1 x 2 = 2

13 x 24 = 312

1357 x 2468 = 3,349,076

13579246801593726048 x 24680135792604815937 = ???

n

How long would it take you to solve this problem?

About n^2 one-digit operations.

- At most n^2 multiplications
- At most n^2 additions (for carries)
- Addition of n different 2n-digit numbers

Can we use divide and conquer to make it faster?

Integer Multiplication

Let's break up a 4-digit integer: $1357 = 13 \cdot 100 + 57$

1357×2468

$= (13 \cdot 100 + 57)(24 \cdot 100 + 68)$

$= (13 \times 24) \cdot 10000 + (13 \times 68 + 57 \times 24) \cdot 100 + (57 \times 68)$

One 4-digit multiplication →
Four 2-digit multiplications

Ist that good?

Integer Multiplication

Let's break up an n-digit integer: $j = a \cdot 10^{n/2} + b$

$j \times k$

$= (a \cdot 10^{n/2} + b)(c \cdot 10^{n/2} + d)$

$= (a \times c) \cdot 10^{n} + (a \times d + b \times c) \cdot 10^{n/2} + (b \times d)$

One n-digit multiplication →
Four (n/2)-digit multiplications

Integer Multiplication

```
algorithm naive_recursive_multiply(j, k):
  Rewrite j as a·10^(n/2) + b
  Rewrite k as c·10^(n/2) + d
  Recursively compute a·c, a·d, b·c, b·d
  Add them up (with shifts) to get j·k
```

Runtime: ???

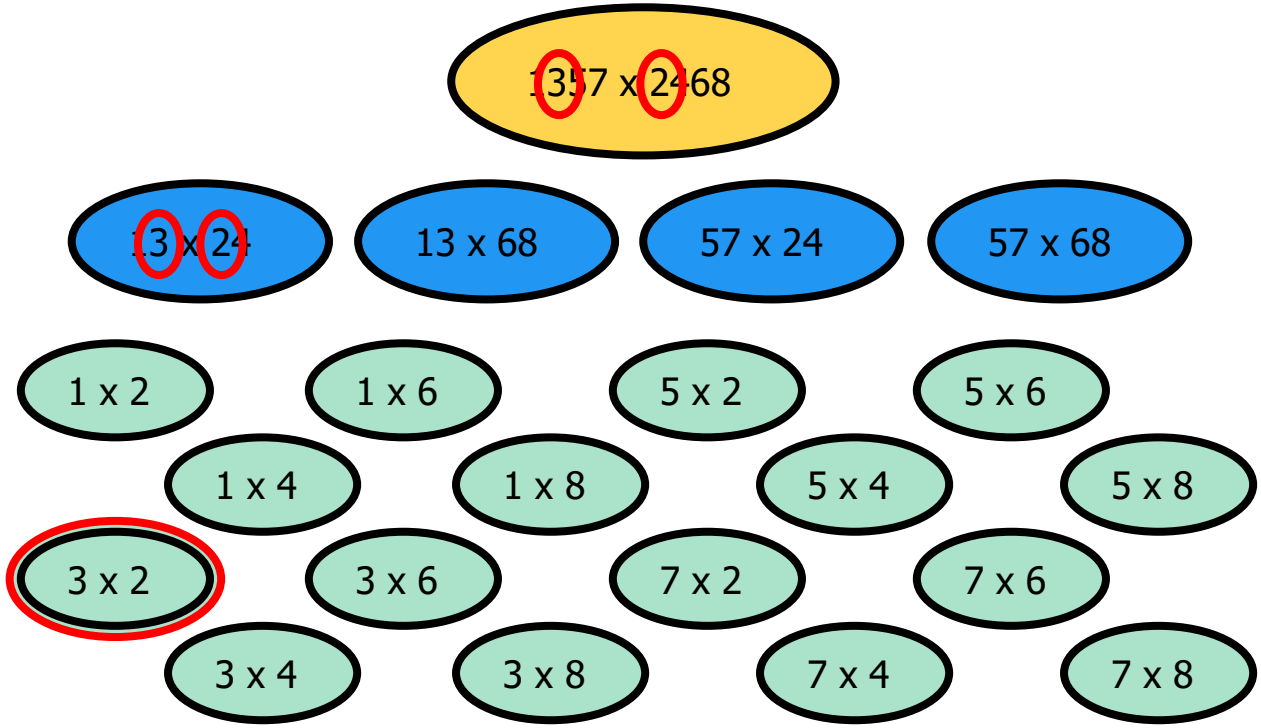
Integer Multiplication

```
algorithm naive_recursive_multiply(j, k):
  Rewrite j as a·10^(n/2) + b
  Rewrite k as c·10^(n/2) + d
  Recursively compute a·c, a·d, b·c, b·d
  Add them up (with shifts) to get j·k
```

Runtime: $O(n^2)$

Analyzing Runtime

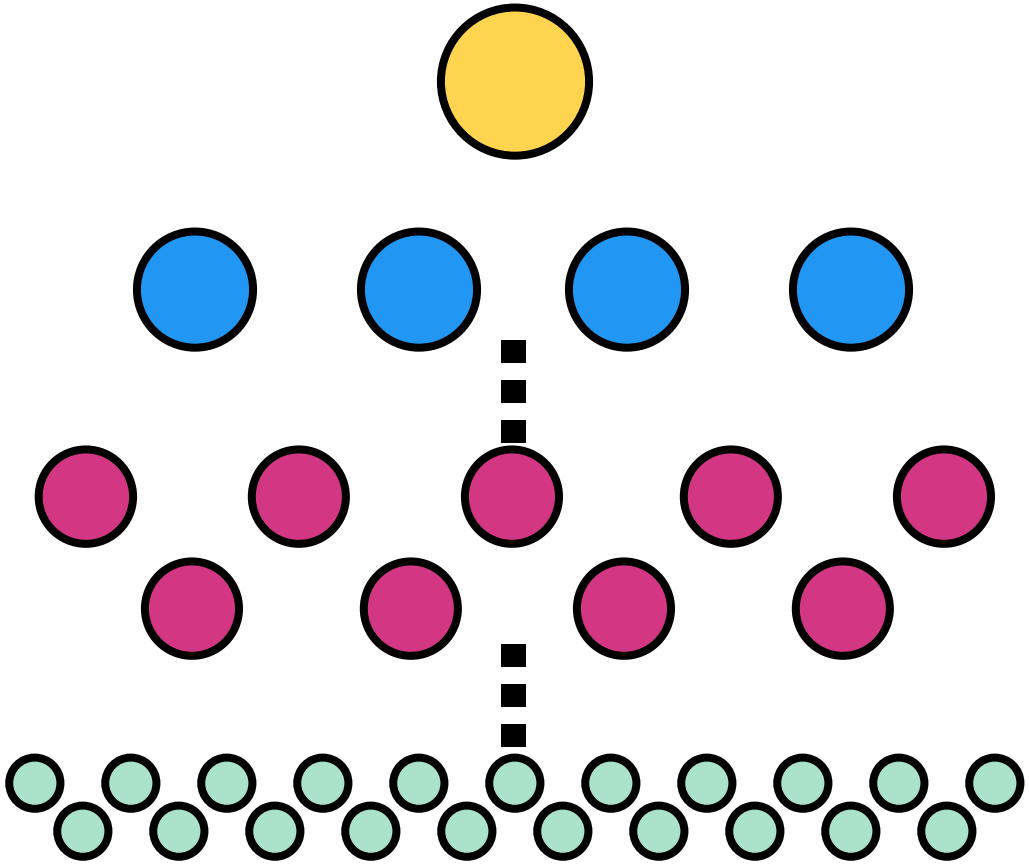
Hm. This is rather suspect ...



Every pair of digits still gets multiplied together separately!

Runtime: $O(n^2)$

Recursion Tree Method



1 problem of size cn — **cn**

4 problems of size cn/2 — **2cn**

4^t problems of size $cn/2^t$ — $\mathbf{2^t cn}$

$4^{log\_2(n)} = n^2$ problems of size c — $\mathbf{cn^2}$

$\log_2 n$ levels

Runtime: $O(n^2)$

For now, take my word that $O(n^2 \log n)$ isn't tight.

Analyzing Runtime

So much work and still $O(n^2)$. This is sad ☹

How to do it better? Any idea?

Karatsuba's Algorithm

Let's break up an n-digit integer: $j = a \cdot 10^{n/2} + b$

j x k

$= (a \cdot 10^{n/2} + b)(c \cdot 10^{n/2} + d)$

$= \underbrace{(a \times c)}_{1} \cdot 10^{n} + \underbrace{(a \times d + b \times c)}_{2,3} \cdot 10^{n/2} + \underbrace{(b \times d)}_{4}$

We needed to spend 4 multiplications: one for each of 1, 2, 3, and 4.

Key insight: 2+3, 1, and 4 are part of the product (a+b)(c+d).

$(a + b)(c + d) = (ad + bc) + (ac) + (bd)$

$(a + b)(c + d) - ac - bd = ad + bc$

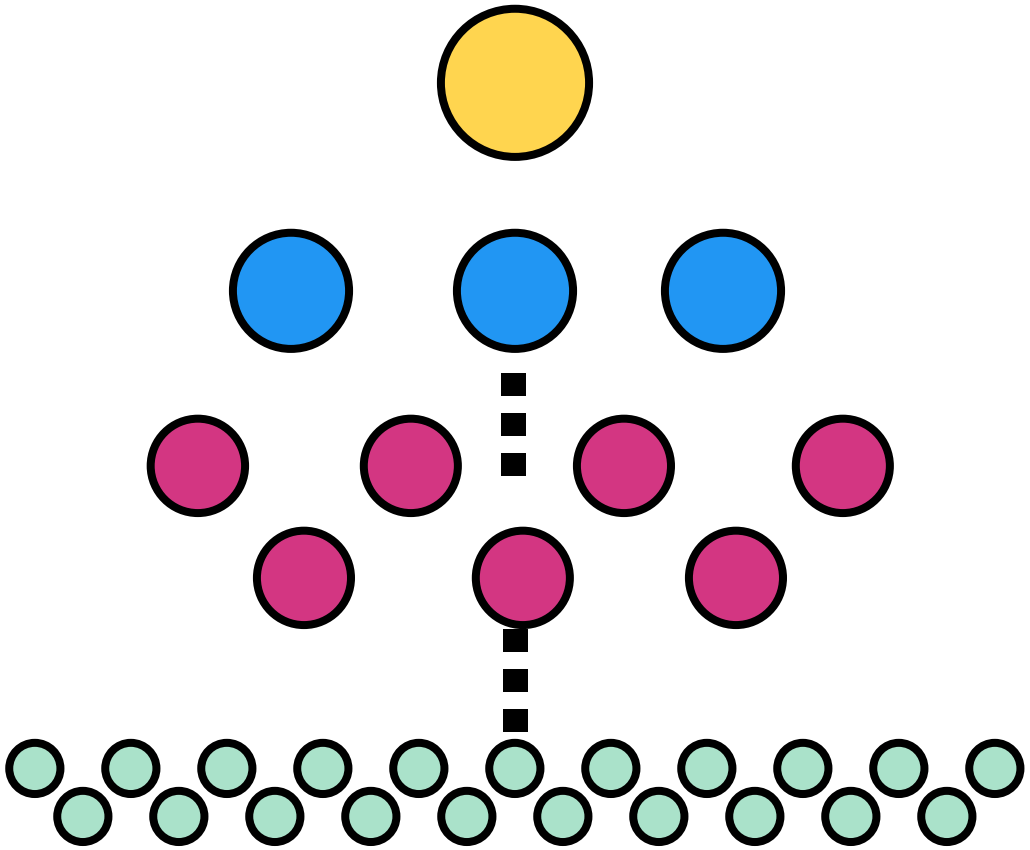
Now, we only need to spend 3 multiplications: one for each of 1 and 4, and a third one for (a+b)(c+d). From these products alone, we can infer 2+3.

Karatsuba's Algorithm

```
algorithm karatsuba_multiply(j, k):
  Rewrite j as a·10^(n/2) + b
  Rewrite k as c·10^(n/2) + d
  Recursively compute a·c, b·d, (a+b)(c+d)
  Let ad+bc = (a+b)(c+d)-ac-bd
  Add them up (with shifts) to get j·k
```

Runtime: $O(n^{\log_2(3)}) = O(n^{1.585})$

Recursion Tree Method



1 problem of size cn — **cn**

3 problems of size cn/2 — **(3/2)cn**

3^t problems of size $cn/2^t$ — **$(3/2)^t cn$**

$3^{log\_2(n)} = n^{log\_2(3)}$ problems of size c — **$cn^{1.585}$**

$log_2 n$ levels

Runtime: $O(n^{1.585})$

For now, take my word that $O(n^{1.585} \log n)$ isn't tight.

Integer Multiplication

$O(n^{1.585})$ runtime of Karatsuba's algorithm is an improvement over $O(n^2)$ runtime of the grade-school algorithm.

A few others outperform Karatsuba's algorithm.

Fun fact 1: The word "algorithm" comes from Al-Khwarizmi, a Persian mathematician who wrote a book (~800 a.d.) about how to multiply Arabic numerals.

Fun fact 2: Karatsuba was a student of Andrej Kolmogorov (a quite famous mathematician). Karatsuba designed that algorithm as part of a student(!) seminar on difficult and unsolvable problems, held by Kolmogorov. After Karatsuba came up with the solution, Kolmogorov was so embarrassed that he cancelled the seminar and did not hold it again.

Solving Recurrences

Solving Recurrences

We've seen the following three recursive algorithms.

`naive_recursive_multiply`

$T(n) = 4T(n/2) + O(n)$
$= O(n^2)$

`karatsuba_multiply`

$T(n) = 3T(n/2) + O(n)$
$= O(n^{log\_2(3)}) = O(n^{1.585})$

`mergesort`

$T(n) = 2T(n/2) + O(n)$
$= O(nlogn)$

What is the pattern???

Master Method

Suppose $T(n) = a \cdot T(n/b) + O(n^d)$.

$$T(n) = \begin{cases} O(n^d \log n) & \text{if } a = b^d \\ O(n^d) & \text{if } a < b^d \\ O(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

where

a is the number of subproblems,

b is the factor by which the input size shrinks, and

d parametrizes the runtime to create the subproblems and merge their solutions.

Master Method

$$T(n) = \begin{cases} O(n^d \log n) & \text{if } a = b^d \\ O(n^d) & \text{if } a < b^d \\ O(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

$$T(n) = a \cdot T(n/b) + O(n^d)$$

We've seen three recursive algorithms.

`naive_recursive_multiply`

$T(n) = 4T(n/2) + O(n)$
$= O(n^2)$

`karatsuba_multiply`

$T(n) = 3T(n/2) + O(n)$
$= O(n^{\log_2(3)}) = O(n^{1.585})$

`mergesort`

$T(n) = 2T(n/2) + O(n)$
$= O(n \log n)$

$a = 4$
$b = 2$
$d = 1$
$a > b^d \rightarrow O(n^{\log_b(a)})$

Wouldn't change if d = 0

$a = 3$
$b = 2$
$d = 1$
$a > b^d \rightarrow O(n^{\log_b(a)})$

Wouldn't change if d = 0

$a = 2$
$b = 2$
$d = 1$
$a = b^d \rightarrow O(n^d \log n)$

Master Method

One can prove the Master Method by writing out a generic proof using a recursion tree [You can try this off-class].

- Draw out the tree.
- Determine the work per level.
- Sum across all levels.

The three cases of the Master Method correspond to whether the recurrence is top heavy, balanced, or bottom heavy.

Summary

- In order to show the runtime complexity of a recursive method, you use:
  - Recursion tree method
  - Proof by induction
  - Master method

Median and Selection

A new problem ...

Motivating question 1

- How many comparisons do we need to compute the smallest element in a list of elements L?

Answer

- Naïve: Sort L and return L[0]
  - Complexity: O(n*logn)
- Faster:
  - Loop over all elements and find minimum
  - Complexity: O(n)

Motivating question 2

- How many comparisons do we need to compute the second smallest element in a list of elements L?

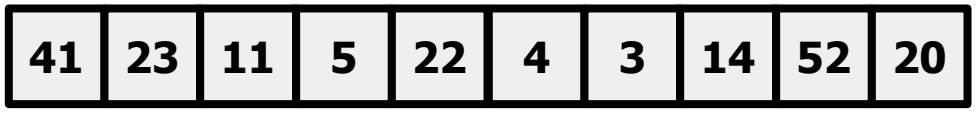
Answer

- Naïve: Sort L and return L[1]
  - Complexity: O(nlogn)
- Faster:
  - Find smallest element first by comparing element neighbours in a tournament style and repeat process recursively for all winning elements
  - Find second smallest element by tournament among all losers, there are only log(n) candidates!
  - Complexity: O(n) … in fact, at most n-1+log(n) comparisons are needed

Now, we will look at a generalization of the problem: Select-k

Select-k Algorithm

In the `select_k` algorithm, we will attempt to return the k^{th} smallest element of an unsorted list of values **A**.

A= 


```
select_k(A,0) => 3
select_k(A,4) => 14
select_k(A,9) => 52
```

```
select_k(A,0) => min(A)
select_k(A,⌈n/2⌉-1) => median(A)
select_k(A,n-1) => max(A)
```

A Slower Select-k Algorithm

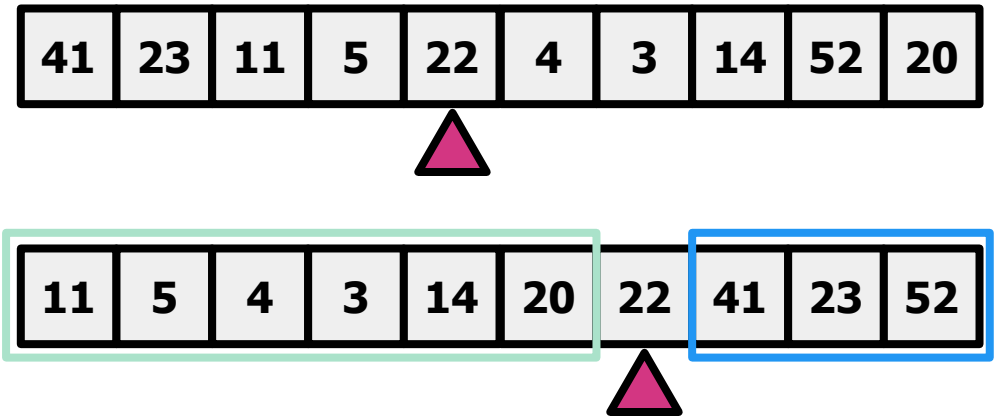
```
algorithm naive_select_k(list A, k):
  A = mergesort(A)
  return A[k]
```

Runtime: O(nlogn)

A better Select-k Algorithm

Main idea: choose a pivot, partition around it, and recurse.

Suppose we call `select_k(A,3)`.



Randomly (for now) choose 22 to be the pivot.

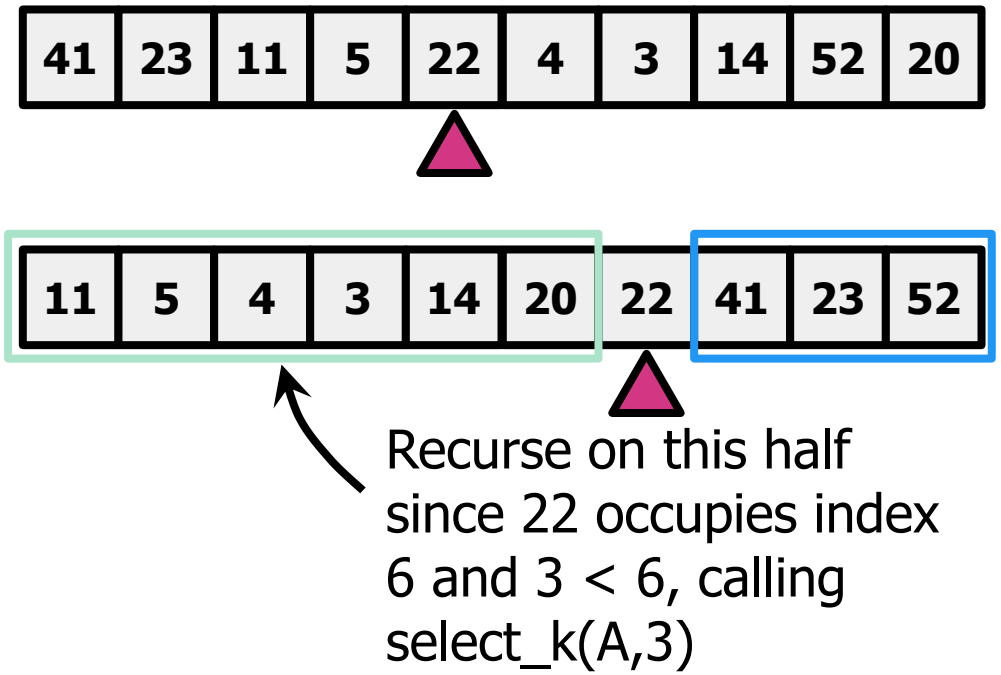
Partition around 22, such that all values to its left are less than it and all values to its right are greater than it.

How to proceed?

A better Select-k Algorithm

Main idea: choose a pivot, partition around it, and recurse.

Suppose we call `select_k(A,3)`.



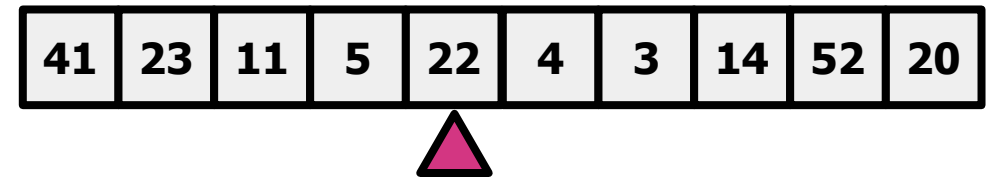
Randomly (for now) choose 22 to be the pivot.

Partition around 22, such that all values to its left are less than it and all values to its right are greater than it.

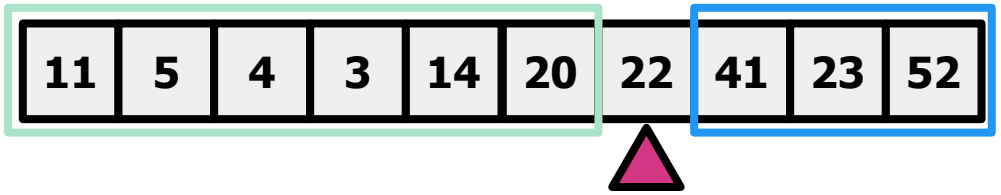
Select-k Algorithm

Main idea: choose a pivot, partition around it, and recurse.

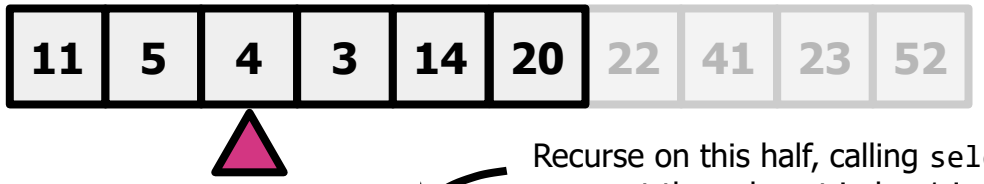
Suppose we call `select_k(A,3)`.



Randomly (for now) choose 22 to be the pivot.

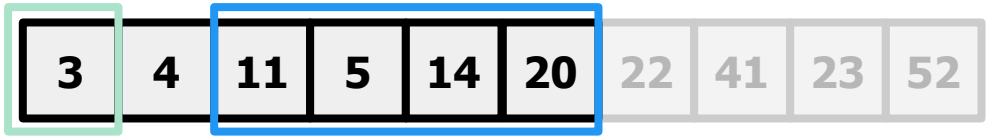


Partition around 22, such that all values to its left are less than it and all values to its right are greater than it.



Randomly (for now) choose 4 to be the pivot.

Recurse on this half, calling `select_k(A[2:],1)` since we want the value at index 1 in the right list.



Partition around 4.

Partitioning

```
algorithm partition(list A, p):
  L, R = []
  for i = 0 to length(A)-1:
    if i == p: continue
    else if A[i] <= A[p]:
      L.append(A[i])
    else if A[i] > A[p]:
      R.append(A[i])
  return L, A[p], R
```

Runtime: $O(n)$

Select-k Algorithm

```
algorithm select_k(list A, k):
  if length(A) == 1: return A[0]
  p = random_choose_pivot(A)
  L, A[p], R = partition(A, p)
  if length(L) == k:
    return A[p]
  else if length(L) > k:
    return select_k(L, k)
  else if length(L) < k:
    return select_k(R, k-length(L)-1)
```

Runtime: $O(n^2)$ ← We'll talk about why this is the case later.

Select-k Algorithm

Feedback from last year: Make sure you get this idea!

```
algorithm select_k(list A, k):
  if length(A) == 1: return A[0]
  p = random_choose_pivot(A)
  L, A[p], R = partition(A, p)
  if length(L) == k:
    return A[p]
  else if length(L) > k:
    return select_k(L, k)
  else if length(L) < k:
    return select_k(R, k-length(L)-1)
```

Runtime: $O(n^2)$ ← We'll talk about why this is the case later.

Thank you very much!