CSE100: Design and Analysis of Algorithms
Lecture 06 – Selection and Median

Feb 3rd 2022

Master Theorem (cont.)

More Recursion, Beyond the Master Theorem

The master theorem (review)

A formula that solves recurrences when all of the sub-problems are the same size

- Suppose that $a \geq 1, b > 1,$ and d are constants (independent of n).
- Suppose $T(n) = a \cdot T\left(\frac{n}{b}\right) + O\left(n^d\right)$. Then

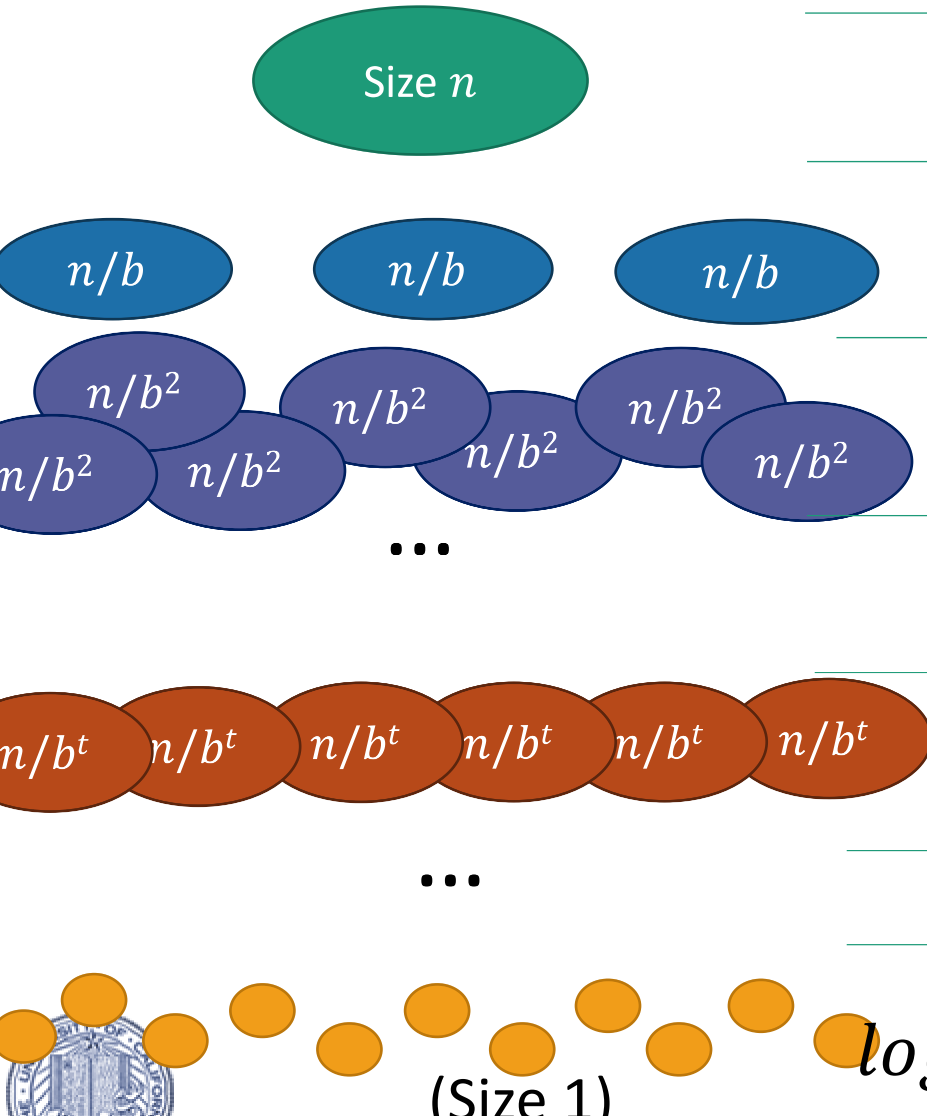
$$T(n) = \begin{cases} O\left(n^d \log(n)\right) & \text{if } a = b^d \\ O\left(n^d\right) & \text{if } a < b^d \\ O\left(n^{\log_b(a)}\right) & \text{if } a > b^d \end{cases}$$

Three parameters:
a : number of subproblems
b : factor by which input size shrinks
d : need to do n^d work to create all the subproblems and combine their solutions.

Many symbols those are….

Recursion tree (review)

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + c \cdot n^d$$



Level	# problems	Size of each problem	Amount of work at this level
0	1	n	$c \cdot n^d$
1	a	n/b	$ac\left(\frac{n}{b}\right)^d$
2	a^2	n/b^2	$a^2c\left(\frac{n}{b^2}\right)^d$
t	a^t	n/b^t	$a^tc\left(\frac{n}{b^t}\right)^d$
...			
$log_b(n)$	$a^{\log_b(n)}$	1	$a^{\log_b(n)}c$ (Let's pretend that the base case isT(1) = c for convenience).

Recursion tree (review)

$T(n) = a \cdot T\left(\frac{n}{b}\right) + c \cdot n^d$

Size n

n/b n/b n/b

n/b

n/b^2

n/b^t

…

(Size 1)

Level	# problems	Size of each problem	Amount of work at this level
0	1	n	$c \cdot n^d$
1	a	n/b	$ac\left(\frac{n}{b}\right)^d$
…			
$log_b(n)$	$a^{\log_b(n)}$	1	$a^{\log_b(n)}c$ (Let's pretend that the base case isT(1) = c for convenience).

Total work is at most:

$$c \cdot n^d \cdot \sum_{t=0}^{log_b(n)} \left(\frac{a}{b^d}\right)^t$$

Today (Part 1)

- How do we measure the runtime of an algorithm?
  - Worst-case analysis
  - Asymptotic Analysis
- Recurrence Relations!
  - How do we calculate the runtime a recursive algorithm?
- The Master Method
  - A useful theorem so we don’t have to answer this question from scratch each time.

Now let's check all the cases

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

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

Case 1: $a = b^d$

- $T(n) = c \cdot n^d \cdot \sum_{t=0}^{log_b(n)} \left(\frac{a}{b^d}\right)^t$

$$= c \cdot n^d \cdot \sum_{t=0}^{log_b(n)} 1$$

$$= c \cdot n^d \cdot (\log_b(n) + 1)$$

$$= c \cdot n^d \cdot \left(\frac{\log(n)}{\log(b)} + 1\right)$$

$$= \Theta\left(n^d \log(n)\right)$$

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

Case 2: $a < b^d$

- $T(n) = c \cdot n^d \cdot \sum_{t=0}^{log_b(n)} \left(\frac{a}{b^d}\right)^t$

Less than 1!

Aside: Geometric sums

- What is $\sum_{t=0}^{N} x^t$?
- You may remember that $\sum_{t=0}^{N} x^t = \frac{x^{N+1}-1}{x-1}$ for $x \neq 1$.
- Morally:

$$x^0 + x^1 + x^2 + x^3 + \cdots + x^N$$

If $0 < x < 1$, this term dominates.

(If $x = 1$, all terms the same)

If $x > 1$, this term dominates.

$$1 \leq \frac{1 - x^{N+1}}{1 - x} \leq \frac{1}{1 - x}$$

(Aka, doesn't depend on N).

$$x^N \leq \frac{x^{N+1}-1}{x - 1} \leq x^N \cdot \left(\frac{x}{x-1}\right)$$

(Aka, $\Theta\left(x^N\right)$ if x is constant and N is growing).

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

Case 2: $a < b^d$

- $T(n) = c \cdot n^d \cdot \sum_{t=0}^{log_b(n)} \left(\frac{a}{b^d}\right)^t$ ← Less than 1!

  $= c \cdot n^d \cdot [\text{some constant}]$

  $= \Theta\left(n^d\right)$

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

Case 3: $a > b^d$

- $T(n) = c \cdot n^d \cdot \sum_{t=0}^{\log_b(n)} \left(\frac{a}{b^d}\right)^t$

Larger than 1!

$$= \Theta\left(n^d \left(\frac{a}{b^d}\right)^{\log_b(n)} \right)$$

Convince yourself that this step is legit!

$$= \Theta\left(n^{\log_b(a)}\right)$$

Now let's check all the cases

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

Even more generally, for T(n) = aT(n/b) + f(n)…

Theorem 3.2 (Master Theorem). *Let $T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$ be a recurrence where $a \geq 1$, $b > 1$. Then,*

- *If $f(n) = O\left(n^{\log_b a - \epsilon}\right)$ for some constant $\epsilon > 0$, $T(n) = \Theta\left(n^{\log_b a}\right)$.*
- *If $f(n) = \Theta\left(n^{\log_b a}\right)$, $T(n) = \Theta\left(n^{\log_b a} \log n\right)$.*
- *If $f(n) = \Omega\left(n^{\log_b a + \epsilon}\right)$ for some constant $\epsilon > 0$ and if $af(n/b) \leq cf(n)$ for $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.*

[From CLRS]

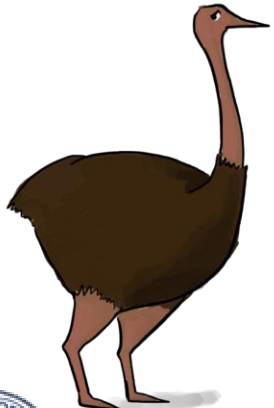


Figure out how to adapt the proof we gave to prove this more general version!

Ollie the Over-Achieving Ostrich

Understanding the Master Theorem

- Let $a \geq 1, b > 1,$ and d be constants.
- Suppose $T(n) = a \cdot T\left(\frac{n}{b}\right) + O\left(n^d\right)$. Then

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

- What do these three cases mean?

The eternal struggle

Branching causes the number of problems to explode!
The most work is at the bottom of the tree!

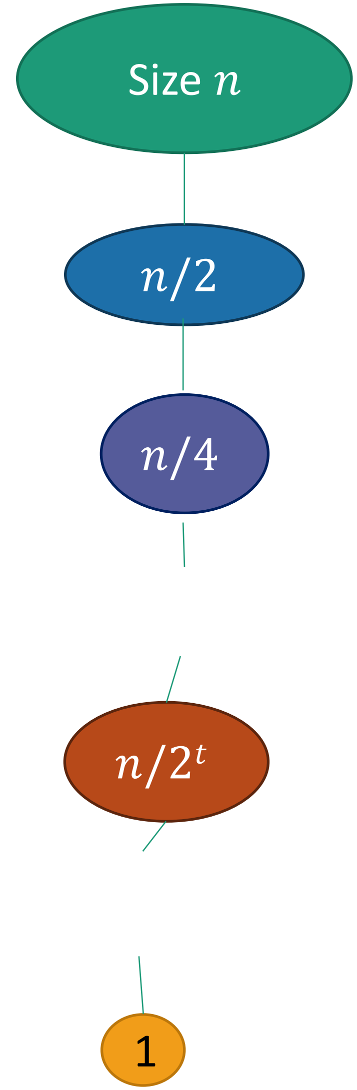
The problems lower in the tree are smaller!
The most work is at the top of the tree!

Consider our three exercise examples

1. $T(n) = T\left(\frac{n}{2}\right) + n$
2. $T(n) = 2 \cdot T\left(\frac{n}{2}\right) + n$
3. $T(n) = 4 \cdot T\left(\frac{n}{2}\right) + n$

First example: tall and skinny tree

1. $T(n) = T\left(\frac{n}{2}\right) + n, \qquad \left(a < b^d\right)$



- The amount of work done at the top (the biggest problem) swamps the amount of work done anywhere else.

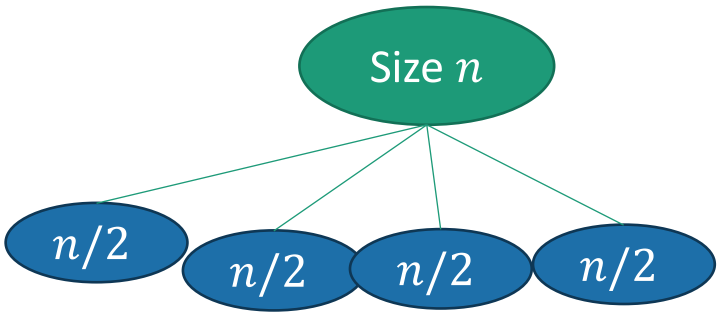
- $T(n) = O(\text{work at top}) = O(n)$

WINNER

Most work at the top of the tree!

Third example: bushy tree

3. $T(n) = 4 \cdot T\left(\frac{n}{2}\right) + n, \qquad (a > b^d)$

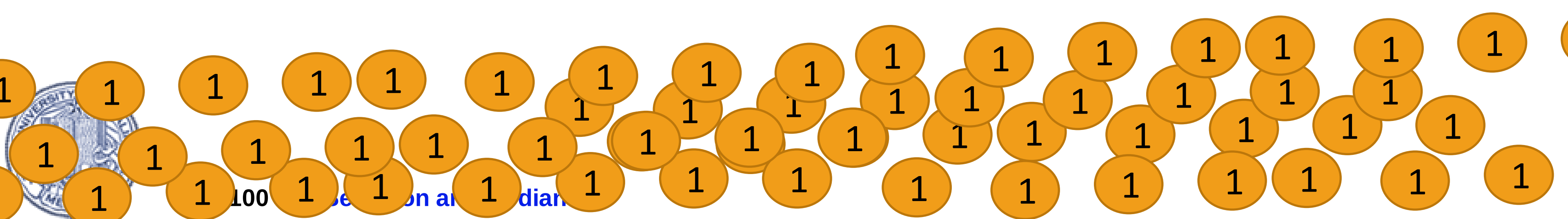


WINNER

Most work at the bottom of the tree!

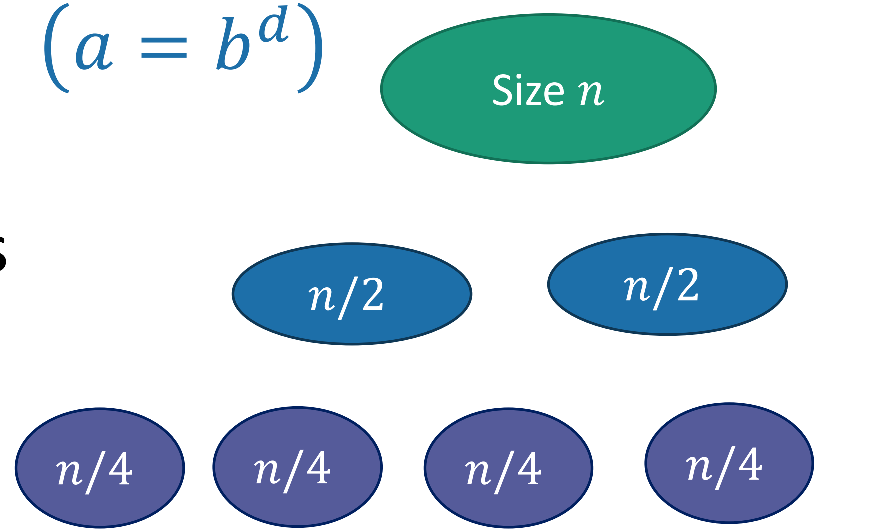
- There are a HUGE number of leaves, and the total work is dominated by the time to do work at these leaves.

- $T(n) = O(work\ at\ bottom) = O(4^{depth\ of\ tree}) = O(n^2)$



Second example: just right

2. $T(n) = 2 \cdot T\left(\frac{n}{2}\right) + n, \qquad \left(a = b^d\right)$



- The branching **just** balances out the amount of work.
- The same amount of work is done at every level.
- $T(n) = (number\ of\ levels) * (work\ per\ level)$
  $= \log(n) * O(n) = O(n log(n))$

What have we learned?

- The “Master Method” makes our lives easier.
- But it’s basically just codifying a calculation we could do from scratch if we wanted to.

Recap

- $O()$ notation makes our lives easier.
- The "Master Method" also make our lives easier.

Next part:

- What if the sub-problems are different sizes?
- And when might that happen?

Some final remarks about the master theorem

- Suppose $T(n) = a \cdot T\left(\frac{n}{b}\right) + O(n^d)$. Then

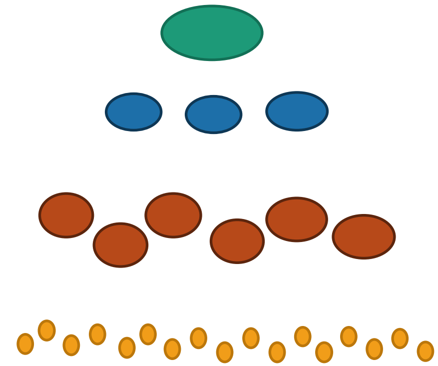
$$T(n) = \begin{cases} O(n^d \log(n)) & \text{if } a = b^d \\ O(n^d) & \text{if } a < b^d \\ O(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

Three parameters:
a : number of subproblems
b : factor by which input size shrinks
d : need to do n^d work to create all the subproblems and combine their solutions.

A powerful theorem it is…

Jedi master Yoda

Some intuition about the cases



Work at level t: $O\left(n^d \left(\frac{a}{b^d}\right)^t\right)$

- Case 1: a = b^d
  - The recursion tree has the same amount of work at every level. (Like MergeSort).
- Case 3: a > b^d
  - The tree branches really quickly compared to work per problem! The bulk of the work is done at the bottom of the tree. (Like Karatsuba).
- Case 2: a < b^d
  - The work done shrinks way faster than we branch new problems. The bulk of the work is done at the root of the tree. (We haven't seen this yet but we will today).

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

More recursion, beyond the Master Theorem

- The Master Theorem only works when all sub-problems are the same size.
- That’s not always the case.
- Today we’ll see an example where the Master Theorem won’t work.
- We’ll use something called the substitution method instead.

I can handle all the recurrence relations that look like $T(n) = a \cdot T\left(\frac{n}{b}\right) + O\left(n^d\right)$. Before this theorem I was but the learner. Now I am the master.

Only a master of evil*, Darth.

*More precisely, only a master of same-size sub-problems…still pretty handy, actually.

The plan for rest of the lecture

1. The Substitution Method.
2. k-SELECT problem
3. k-SELECT solution
4. Return of the Substitution Method.

Why is this useful?
The substitution method

- Recursion trees can get pretty messy here, since we have a recurrence relation that doesn't nicely break up our big problem into sub-problems of the same size.
- Instead, we will try to:
  - Make a guess
  - Check using an inductive argument
- This is called the substitution method.

The Substitution Method

- Another way to solve recurrence relations.
- More general than the master method.

- Step 1: Generate a guess at the correct answer.
- Step 2: Try to prove that your guess is correct.
- (Step 3: Profit.)

Substitution method

work to call/create recursive sub-problems and combine them back

- Suppose that $T(n) \leq c \cdot f(n) + \sum_{i=1}^{r} T(n_i)$

Work in r different sub-problems, which might have different sizes.

- Let's guess the solution is

$$T(n) \leq \begin{cases} d \cdot g(n_0) & if \quad n \leq n_0 \\ d \cdot g(n) & if \quad n > n_0 \end{cases} \qquad (*)$$

- (aka, guessing $T(n) = O(g(n))$)
- We'll prove this by induction, with the inductive hypothesis (*) for all smaller n's.

The Substitution Method

first example

- Let's return to:

  $T(n) = 2 \cdot T\left(\frac{n}{2}\right) + n$, with $T(0) = 0, T(1) = 1$.

- The Master Method says $T(n) = O(n \log(n))$.
- We will prove this via the Substitution Method.

Step 1: Guess the answer

- $T(n) = 2 \cdot T\left(\frac{n}{2}\right) + n$

Expand $T\left(\frac{n}{2}\right)$

- $T(n) = 2 \cdot \left(2 \cdot T\left(\frac{n}{4}\right) + \frac{n}{2}\right) + n$

Simplify

- $T(n) = 4 \cdot T\left(\frac{n}{4}\right) + 2n$

Expand $T\left(\frac{n}{4}\right)$

- $T(n) = 4 \cdot \left(2 \cdot T\left(\frac{n}{8}\right) + \frac{n}{4}\right) + 2n$

Simplify

- $T(n) = 8 \cdot T\left(\frac{n}{8}\right) + 3n$
- …

You can guess the answer however you want: meta-reasoning, a little bird told you, wishful thinking, etc. One useful way is to try to "unroll" the recursion, like we're doing here.

Guessing the pattern: $T(n) = 2^t \cdot T\left(\frac{n}{2^t}\right) + t \cdot n$

Plug in $t = \log(n)$, and get

$$T(n) = n \cdot T(1) + \log(n) \cdot n = n(\log(n) + 1)$$

$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + n$, with $T(1) = 1$.

Step 2: Prove the guess is correct.

- Inductive Hyp. (n): $T(j) = j(\log(j) + 1)$ for all $1 \le j \le n$.
- Base Case ($n = 1$): $T(1) = 1 = 1 \cdot (\log(1) + 1)$
- Inductive Step:
  - Assume Inductive Hyp. for $n = k - 1$:
    - Suppose that $T(j) = j(\log(j) + 1)$ for all $1 \le j \le k - 1$.
  - $T(k) = 2 \cdot T\left(\frac{k}{2}\right) + k$ by definition
  - $T(k) = 2 \cdot \left(\frac{k}{2}\left(\log\left(\frac{k}{2}\right) + 1\right)\right) + k$ by induction.
  - $T(k) = k(\log(k) + 1)$ by simplifying.
  - So Inductive Hypothesis holds for $n = k$.
- Conclusion: For all $n \ge 1$, $T(n) = n(\log(n) + 1)$

We just replaced the "n" in the statement of the inductive hypothesis with an "k-1" to get the I.H. for k-1.

We're being sloppy here about floors and ceilings…what would you need to do to be less sloppy?

Step 3: Profit

- Pretend like you never did Step 1, and just write down:

- *Theorem:* $T(n) = O(n \log(n))$
- *Proof:* *[Whatever you wrote in Step 2]*

What have we learned?

- The substitution method is a different way of solving recurrence relations.

- Step 1: Guess the answer.
- Step 2: Prove your guess is correct.
- Step 3: Profit.

- We'll get more practice with the substitution method next lecture!

Another example (if time)

(If not time, that's okay; we'll see these ideas later)

- $T(n) = 2 \cdot T\left(\frac{n}{2}\right) + 32 \cdot n$
- $T(2) = 2$

- Step 1: Guess: $O(n\log(n))$ (divine inspiration).
- But I don't have such a precise guess about the form for the $O(n\log(n))$ …
    - That is, what's the leading constant?
- Can I still do Step 2?

Step 2: Prove it, working backwards to figure out the constant

- Guess: $T(n) \leq C \cdot n \log(n)$ for some constant C TBD.
- Inductive Hypothesis: $T(j) \leq C \cdot j \log(j)$ for $2 \leq j \leq n$
- Base case: $T(2) = 2 \leq C \cdot 2 \log(2)$ as long as $C \geq 1$
- Inductive Step:

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + 32 \cdot n$$
$$T(2) = 2$$

Inductive step

- Assume that the inductive hypothesis holds for $n = k-1$.
- $T(k) = 2T\left(\frac{k}{2}\right) + 32k$
- $\qquad \le 2C\frac{k}{2}\log\left(\frac{k}{2}\right) + 32k$
- $\qquad = k(C \cdot \log(k) + 32 - C)$
- $\qquad \le k(C \cdot \log(k))$ as long as $C \ge 32$.
- Then the inductive hypothesis holds for $n = k$.

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + 32 \cdot n$$
$$T(2) = 2$$

Step 2: Prove it, working backwards to figure out the constant

- Guess: $T(n) \leq C \cdot n \log(n)$ for some constant C TBD.
- Inductive Hypothesis: $T(j) \leq C \cdot j \log(j)$ for $2 \leq j \leq n$
- Base case: $T(2) = 2 \leq C \cdot 2 \log(2)$ as long as $C \geq 1$
- Inductive step: Works as long as $C \geq 32$
  - So choose $C = 32$.
- Conclusion: $T(n) \leq 32 \cdot n \log(n)$

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + 32 \cdot n$$
$$T(2) = 2$$

Step 3: Profit.

- *Theorem:* $T(n) = O(n \log(n))$
- *Proof:*
  - Inductive Hypothesis: $T(j) \leq 32 \cdot j \log(j)$ for $2 \leq j \leq n$
  - Base case: $T(2) = 2 \leq 32 \cdot 2 \log(2)$ is true.
  - Inductive step:
    - Assume Inductive Hyp. for $n = k - 1$.
    - $T(k) = 2T\left(\frac{k}{2}\right) + 32k$ By the def. of $T(k)$
    - $\leq 2 \cdot 32 \cdot \frac{k}{2} \log\left(\frac{k}{2}\right) + 32k$ By induction
    - $= k(32 \cdot \log(k) + 32 - 32)$
    - $= 32 \cdot k \log(k)$
    - This establishes inductive hyp. for $n = k$.
  - Conclusion: $T(n) \leq 32 \cdot n \log(n)$ for all $n \geq 2$.

Aside:

The form of the inductive hypothesis

- In the previous examples, we had an inductive hypothesis of the form:

$$T(j) \leq 32 \cdot j \log(j) \text{ for } 2 \leq j \leq n$$

  - The reason it was written like that is because that's what it should be if I'm doing "standard" induction. That is, the inductive step is: assuming that the inductive hypothesis holds for $k-1$, show that it holds for k.

- However, if one uses strong induction, it's fine to use an inductive hypothesis of the form:

$$T(n) \leq 32 \cdot n \log(n)$$

  - In this case, the inductive step would be: assuming that the inductive hypothesis holds for all $2 \leq j < k$, show that it holds for k.

- Both ways are totally fine.

Solving Recurrence Relations

- A **recurrence relation** expresses $T(n)$ in terms of $T(\text{less than } n)$
- For example, $T(n) = 2 \cdot T\left(\frac{n}{2}\right) + 11 \cdot n$
- Two methods of solution:
  1. Master Theorem (aka, generalized “tree method”)
  2. Substitution method (aka, guess and check)

What have we learned?

- The substitution method can work when the master theorem doesn't.
  - For example with different-sized sub-problems.

- Step 1: generate a guess
  - Throw the kitchen sink at it.
- Step 2: try to prove that your guess is correct
  - You may have to leave some constants unspecified till the end – then see what they need to be for the proof to work!!
- Step 3: profit
  - Pretend you didn't do Steps 1 and 2 and write down a nice proof.

The Plan

1. More practice with the Substitution Method.
2. k-SELECT problem
3. k-SELECT solution
4. Return of the Substitution Method.

The k-SELECT problem

For today, assume all arrays have distinct elements.

A is an array of size n, k is in {1,…,n}

- SELECT(A, k):
  - Return the k'th smallest element of A.

7	4	3	8	1	5	9	14

- SELECT(A, 1) = 1
- SELECT(A, 2) = 3
- SELECT(A, 3) = 4
- SELECT(A, 8) = 14

- SELECT(A, 1) = MIN(A)
- SELECT(A, n/2) = MEDIAN(A)
- SELECT(A, n) = MAX(A)

Being sloppy about floors and ceilings!

Note that the definition of Select is 1-indexed…

An O(nlog(n))-time algorithm

- SELECT(A, k):
  - A = MergeSort(A)
  - **return** A[k-1]

It's k-1 and not k since my pseudocode is 0-indexed and the problem is 1-indexed…

- Running time is O(n log(n)).
- So that's the benchmark….

Can we do better?

We're hoping to get O(n)

Show that you can't do better than O(n).

Goal: An O(n)-time algorithm

- Let's start with MIN(A) aka SELECT(A, 1).
- MIN(A):
  - **For** i=1, ..., n:
    - If A[i] < ret:
      - ret = A[i]
  - **Return** ret

This stuff is O(1)

This loop runs O(n) times

- Time O(n). Yay!

How about SELECT(A,2)?

- SELECT2(A):
  - ret = ∞
  - minSoFar = ∞
  - **For** i=1, ..., n:
    - If A[i] < ret and A[i] < minSoFar:
      - ret = minSoFar
      - minSoFar = A[i]
    - Else if A[i] < ret and A[i] >= minSoFar:
      - ret = A[i]
  - **Return** ret

(The actual algorithm here is not very important because this won't end up being a very good idea...)

Still O(n)

SO FAR SO GOOD.

SELECT(A, n/2) aka MEDIAN(A)?

- MEDIAN(A):
  - ret = ∞
  - minSoFar = ∞
  - secondMinSoFar = ∞
  - thirdMinSoFar = ∞
  - fourthMinSoFar = ∞
  - ….

- This is not a good idea for large k (like n/2 or n).
- Basically this is just going to turn into something like INSERTIONSORT…and that was $O(n^2)$.

The Plan

1. More practice with the Substitution Method.
2. k-SELECT problem
3. k-SELECT solution
4. Return of the Substitution Method.

Idea: divide and conquer!

Say we want to find SELECT(A, k)

9	8	3	6	1	4	2

Idea: divide and conquer!

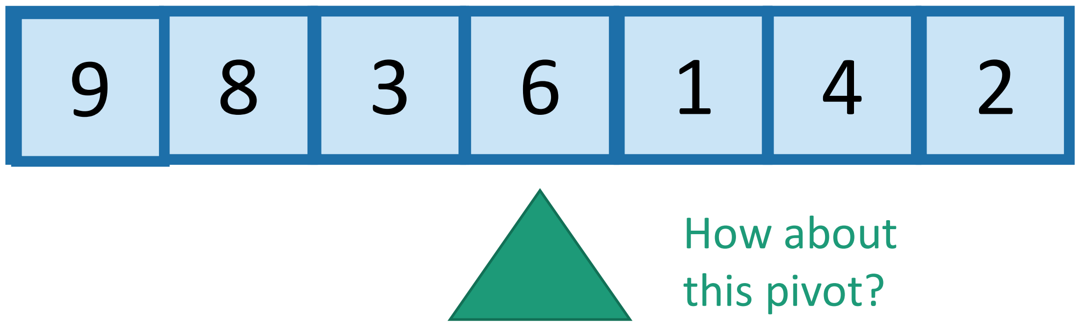
Say we want to find SELECT(A, k)

9	8	3	6	1	4	2

First, pick a "pivot." We'll see how to do this later.

Idea: divide and conquer!

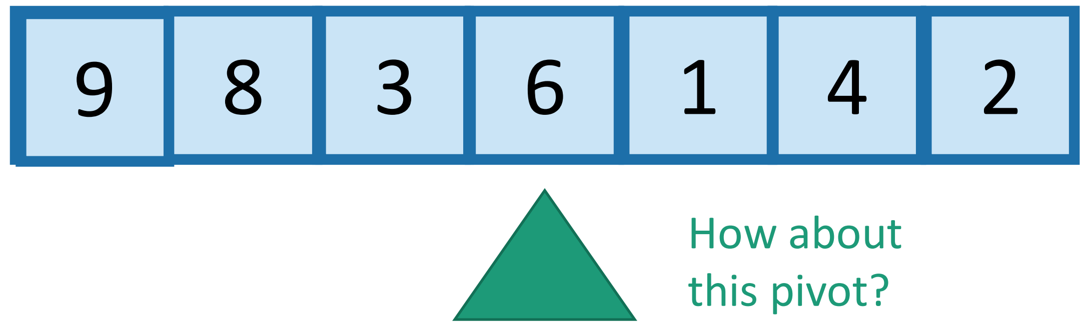
Say we want to find SELECT(A, k)



First, pick a "pivot." We'll see how to do this later.

Idea: divide and conquer!

Say we want to find SELECT(A, k)



First, pick a "pivot." We'll see how to do this later.

Next, partition the array into "bigger than 6" or "less than 6"

Idea: divide and conquer!

Say we want to find SELECT(A, k)

9	8	3	6	1	4	2

How about this pivot?

First, pick a "pivot." We'll see how to do this later.

Next, partition the array into "bigger than 6" or "less than 6"

L = array with things smaller than A[pivot]

R = array with things larger than A[pivot]

Idea: divide and conquer!

Say we want to find SELECT(A, k)

9	8	3	6	1	4	2

How about this pivot?

First, pick a "pivot." We'll see how to do this later.

Next, partition the array into "bigger than 6" or "less than 6"

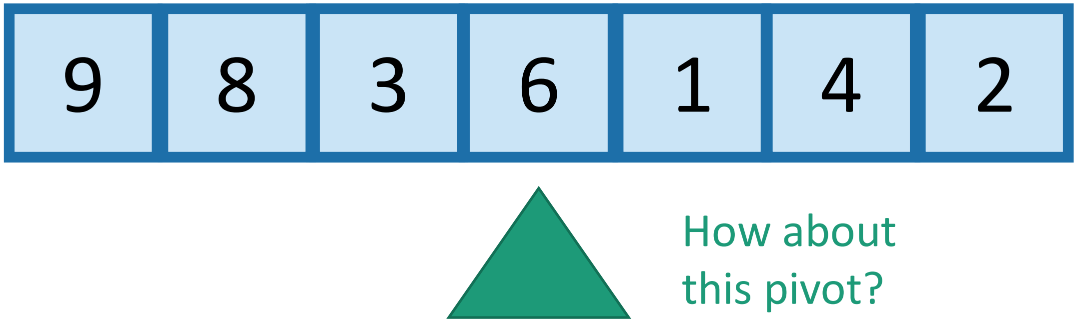
9

L = array with things smaller than A[pivot]

R = array with things larger than A[pivot]

Idea: divide and conquer!

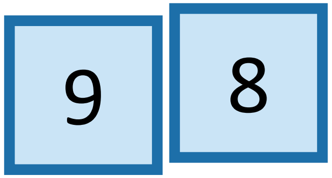
Say we want to find SELECT(A, k)



How about this pivot?

First, pick a "pivot." We'll see how to do this later.

Next, partition the array into "bigger than 6" or "less than 6"

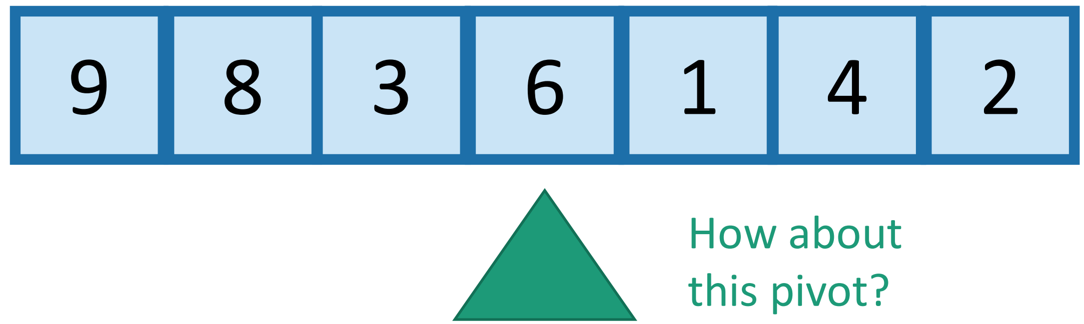


L = array with things smaller than A[pivot]

R = array with things larger than A[pivot]

Idea: divide and conquer!

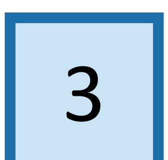
Say we want to find SELECT(A, k)



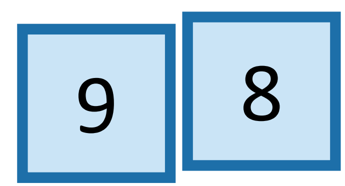
How about this pivot?

First, pick a "pivot." We'll see how to do this later.

Next, partition the array into "bigger than 6" or "less than 6"



L = array with things smaller than A[pivot]



R = array with things larger than A[pivot]

Idea: divide and conquer!

Say we want to find SELECT(A, k)

9 8 3 6 1 4 2

How about this pivot?

First, pick a "pivot." We'll see how to do this later.

Next, partition the array into "bigger than 6" or "less than 6"

3 1

L = array with things smaller than A[pivot]

9 8

R = array with things larger than A[pivot]

Idea: divide and conquer!

Say we want to find SELECT(A, k)

9	8	3	6	1	4	2

How about this pivot?

First, pick a "pivot." We'll see how to do this later.

Next, partition the array into "bigger than 6" or "less than 6"

3	1	4

L = array with things smaller than A[pivot]

9	8

R = array with things larger than A[pivot]

Idea: divide and conquer!

Say we want to find SELECT(A, k)

9	8	3	6	1	4	2

How about this pivot?

First, pick a "pivot." We'll see how to do this later.

Next, partition the array into "bigger than 6" or "less than 6"

3	1	4	2

L = array with things smaller than A[pivot]

9	8

R = array with things larger than A[pivot]

Idea: divide and conquer!

Say we want to find SELECT(A, k)

9	8	3	6	1	4	2

How about this pivot?

First, pick a "pivot." We'll see how to do this later.

Next, partition the array into "bigger than 6" or "less than 6"

This PARTITION step takes time O(n). (Notice that we don't sort each half).

3	1	4	2

L = array with things smaller than A[pivot]

9	8

R = array with things larger than A[pivot]

Idea: divide and conquer!

Say we want to find SELECT(A, k)

First, pick a "pivot." We'll see how to do this later.

6

How about this pivot?

Next, partition the array into "bigger than 6" or "less than 6"

This PARTITION step takes time O(n). (Notice that we don't sort each half).

3	1	4	2

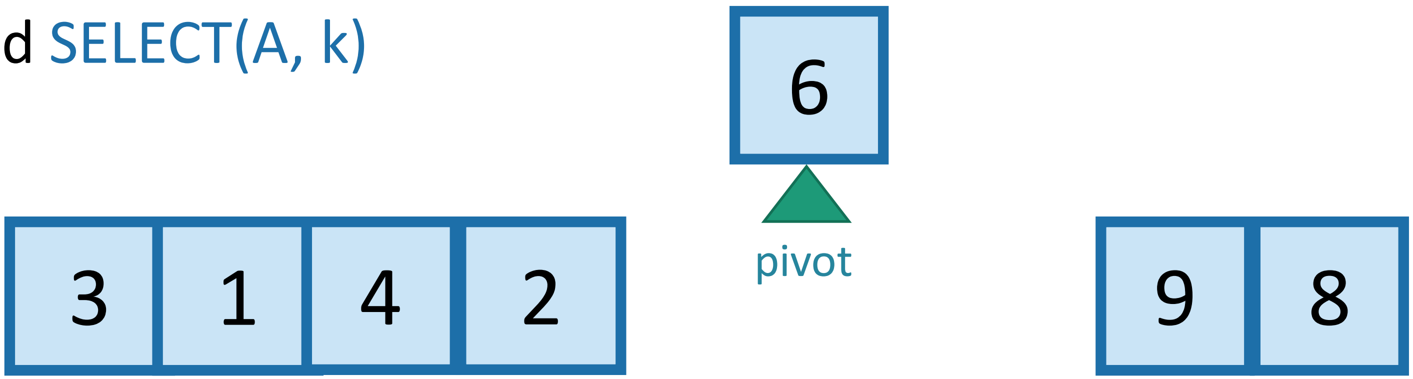
L = array with things smaller than A[pivot]

9	8

R = array with things larger than A[pivot]

Idea continued…

Say we want to find SELECT(A, k)

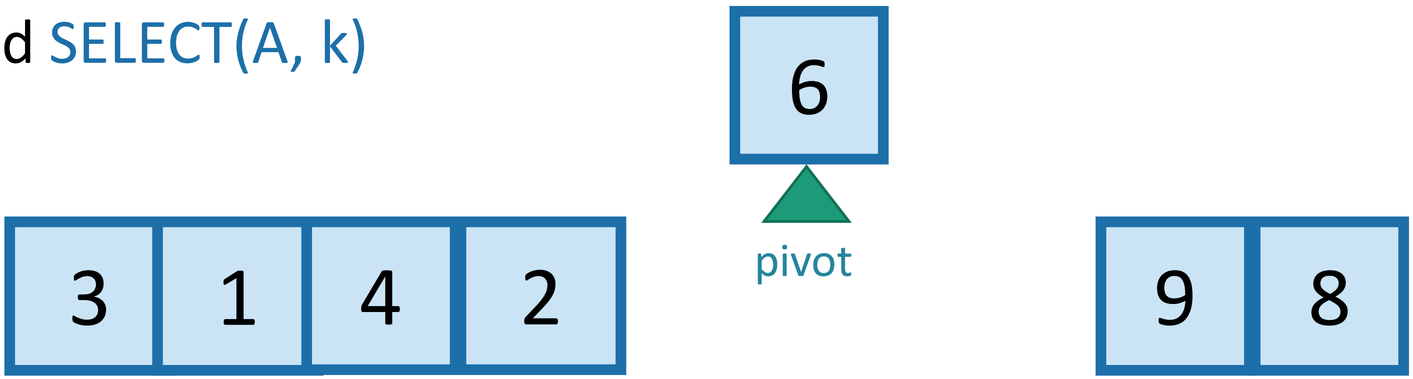


L = array with things smaller than A[pivot]

R = array with things larger than A[pivot]

Idea continued…

Say we want to
find SELECT(A, k)



L = array with things
smaller than A[pivot]

R = array with things
larger than A[pivot]

- If k = 5 = len(L) + 1:
  - We should return A[pivot]
- If k < 5:
  - We should return SELECT(L, k)
- If k > 5:
  - We should return SELECT(R, k – 5)

This suggests a
recursive algorithm

(still need to figure out
how to pick the pivot…)