SIT320 Advanced Algorithms – Task 10 Algorithm Analysis

[redacted]

Table of Contents

GitHub Link:

Learning Summary

Module 10 explored Algorithmic complexity, asymptotic analysis, substitution method, unbalanced recurrences and Akra-Bazzi method, The Master Theorem, Recursion Tree method and The Select(A, k) Problem. I found this module to be challenging as it extended my learning on algorithmic complexities. In previous units I learnt why data structures were important (to efficiently store/ retrieve data to make informed decisions). And I was award of sorting algorithms like Merge sort, Insertion Sort, Heap Sort etc and how they arrange data. This knowledge I already had enabled me to make better informed choices for which algorithm to use for worst case performance.

It was good to review asymptotic analysis, which is a technique used to analyse the performance of algorithms, this helped to strengthen my understanding of divide and conquer methods and remind me how this can lead to unbalanced divisions of work.
I didn't realise Merge sort is efficient in time whereas Insertion sort is efficient in space. I was excited to learn more about recurrence relations and how they describe a functions value based on smaller inputs. I believe this unit wanted me to understand that you need a deep understanding of algorithmic complexity using asymptotic analysis for solving recurrences.

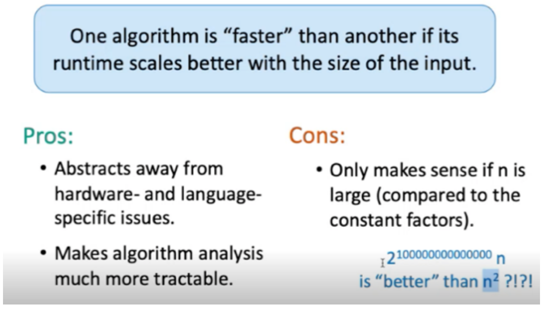
Substitution method, which is the guaranteed approach to finding the right result when analysing algorithmic complexity was also explored. I enjoyed learning how general substitution (induction) a loop invariant is maintained, and the proof can be found in specific steps. The unbalanced recurrences for Akra-Bazzi took a significant amount of time to understand, I used external resources for this content listed in my below references. The Akra-Bazzi method is a guaranteed solution of a wide range of problems and is used to solve unbalanced recurrences with two steps.

The Master Theorem was a new concept for me, but through testing many recurrences I learnt that all subproblems need to be an equal size and relies on specific parameters. I did like that there was an alternative formulation for handling 'd' parameter which would make the application of this theorem easier for identifying other recurrence relationships. The Recursion Tree Method represents the cost of subproblems (nodes in a tree) and this calculated for per-level cost. I didn't know this method was used to generate initial guesses. The Select(A, k) problem can only be used when sorting small lists to find specific elements in an unsorted list. I didn’t know that divide and conquer methods (Quick Sort) could lead to recurrences from this. However, through the activities I found Median of Medians when used for pivot selection allows a more balanced divide and conquer approach. I hope to include some of these methods in the next tasks as I believe these concepts are crucial to this unit and my future career.

References, included at bottom of report.

Algorithmic Complexity

- Data structures are important to make *informed decisions* [1], [2]
    - An agreed arrangement, so that’s it’s easy to store as well as find the desired items
- Algorithms are created to store and retrieve data in the most efficient way
    - Data can be stored in cloud, database, data center etc.
- Sorting algorithms
    - Merge sort, insertion sort, quick sort, heap sort, bucket sort etc.
- Algorithm Complexity
    - Efficiency of storing and retrieving data from a data source
    - Data can be stored in memory, database, SQL or NoSQL etc.
- Which algorithm is best?
    - Big O Notation to categories an algorithm
    - Always present worst case



One algorithm is “faster” than another if its runtime scales better with the size of the input.

Pros:

- Abstracts away from hardware- and language-specific issues.
- Makes algorithm analysis much more tractable.

Cons:

- Only makes sense if n is large (compared to the constant factors).

$2^{100000000000000}$ n is “better” than n^2 ?!?!

- $O(n)$ Complexity
    - n => number of items
    - At worst case, it will have to search ‘n’ items to find the desired one
    - Starting at front of array ‘n’ items and find the last ‘n -1’ item in array
    - n items = n steps

- On^2 / $O(n \cdot n)$ Complexity
    - n => number of items
    - Worst case, it will have to perform $n \cdot n$ steps to find the desired one

n = 1 stack of shirts (3 x shirts), n = 1 stack of pants (3 x pants), match each item

$$3 + 3 + 3 = 9$$
$$3 * 3 = 0$$
$$3^2 = 9$$

- $O(n \cdot m)$ Complexity
    - n => number of items
    - Worst case, it will have to perform $n \cdot m$ steps to find the desired one

n = 1 stack of shirts (3 x shirts), m = 1 stack of pants (2 x pants), match each item

$$2 + 2 + 2 = 6$$
$$3 * 2 = 6$$

- $O(log\ n)$ Complexity
    - n => number of items

   - Worst case, it will have to perform $log\ n$ steps to find the desired one

Example: As your number of items grow (very slowly), your number of steps grow, BUT it is not exponential it is *logarithmic*

No matter how big the dictionary is, you can find a word in 2 or 3 steps, must be stored in a sorted order. Binary search tree (sorted elements)

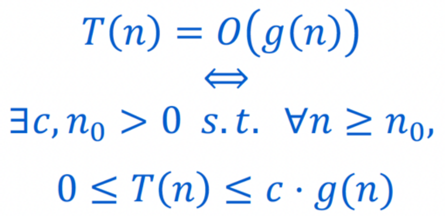
$$log\ (10) = 1 \text{ (steps)}$$
$$log\ (100) = 2 \text{ (steps)}$$
$$log\ (1000) = 3 \text{ (steps)}$$

- $O(n\ log\ n)$ Complexity
   - n => number of items
   - Worst case, it will have to perform $n \cdot log\ (n)$ steps to find the desired one

n = 1 stack of shirts (3 x shirts), $log\ (n)$ = Items in store, match each item

Go to a store to find the match. Even if the store has hundred items, we can find items in a few steps as they are in sorted order (number, colour etc)

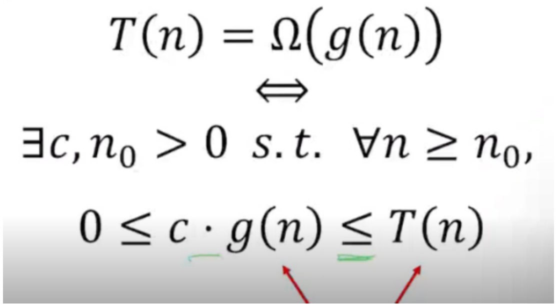
$$3\ * (2 - 5\ matches)$$
$$n * log\ n$$



$$T(n) = O\big(g(n)\big)$$
$$\Leftrightarrow$$
$$\exists c, n_0 > 0\ \ s.t.\ \ \forall n \geq n_0,$$
$$0 \leq T(n) \leq c \cdot g(n)$$

O(…) means a UPPER bound

- Is basically the UPPER BOUND
- $T(n)$ is the runtime: positive and increasing in n
- $T(n)\ is\ O(g(n))$ if g(n) grows at **least** as fast as $T(n)$ as n gets large!



$$T(n) = \Omega\big(g(n)\big)$$
$$\Leftrightarrow$$
$$\exists c, n_0 > 0\ \ s.t.\ \ \forall n \geq n_0,$$
$$0 \leq c \cdot g(n) \leq T(n)$$

Ω(…) means a LOWER bound

- $T(n)\ is\ \ \Omega(g(n))$ if g(n) grows at **most** as fast as $T(n)$ as n gets large!

θ(…) means both (lower/ upper) bound

- $T(n)\ is\ \ \theta(g(n))$ if,

$$T(n) = O\big(g(n)\big)$$
and
$$T(n) = \ \Omega(\mathrm{g(n)})$$

Asymptotic Analysis

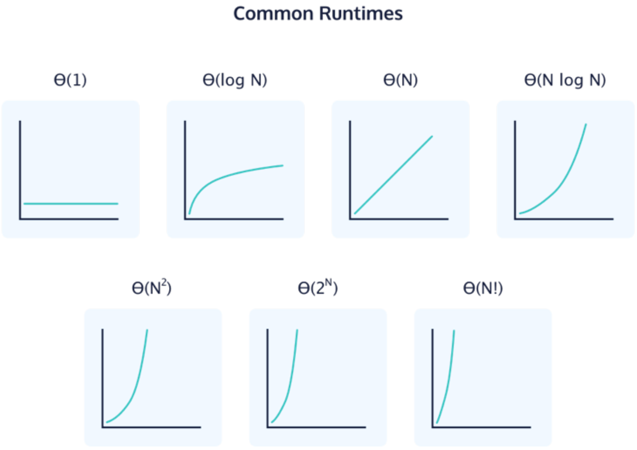
- Is used to describe the running time of an algorithm, how much time an algorithm takes with a given input n [3]

1. *Big O* – worst case running time
2. *Big Theta θ*
3. *Big Omega Ω – best case running time*

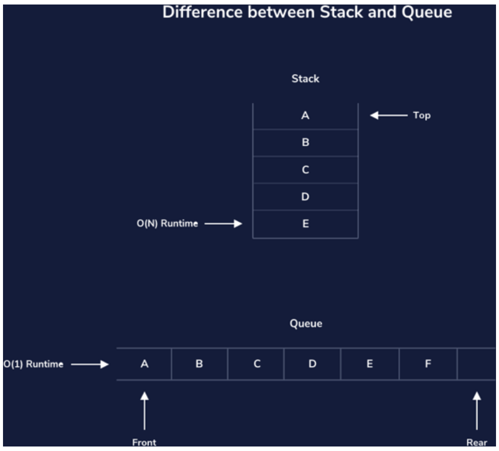
- Compute the *Big O* of an algorithm by counting the number of iterations the algorithm *always* takes with an input of n.
- This loop will always iterate n times for a list of n

```
        for each item in list:
            print item
```

- If algorithm contains many parts – describe runtime based on the *Slowest* part of program
- Common runtimes –
- Constant $O(1)$
- Logarithmic $O(\log n)$
- Linear $O(n)$
- Polynomial $O(n^2)$
- Exponential $O(2^n)$
- Factorial $O(n!)$

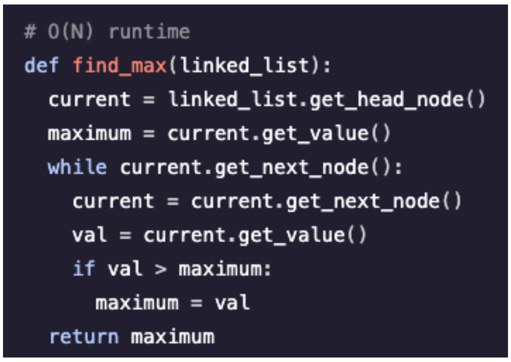


- Speed of algorithm measured using while loop
  - Loop can be used to count the number of iterations it takes a function to complete
- Queue data structure based on 'First in *First* Out Order' taking $O(1)$runtime Constant
  - To retrieve the first item in a queue
- Stack data structure is based on 'First in *Last* out order' taking $O(n)$runtime Linear
  - To retrieve the first value in a stack, all the way at the bottom!



Max Value Search in List

  - For locating the maximum value in a list of size n is $O(n)$ Linear
    - Due to the entire list of n items being traversed



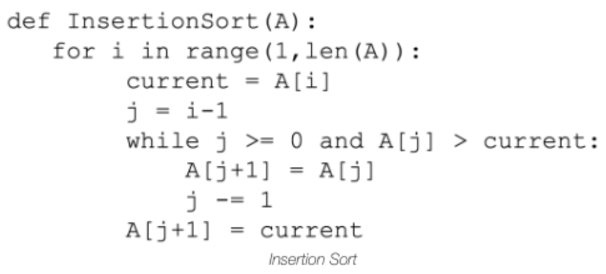
```
# O(N) runtime
def find_max(linked_list):
  current = linked_list.get_head_node()
  maximum = current.get_value()
  while current.get_next_node():
    current = current.get_next_node()
    val = current.get_value()
    if val > maximum:
      maximum = val
  return maximum
```

Bubble Sort

- Simplest sorting algorithm for a list
- Every item in list, compares it with its subsequent neighbor and swaps them if they are in *descending order*
- Each pass of swap takes $O(n)$, there are n items in list, taking $O(n \cdot n)$ swaps
- *Big O* = $O(n^2)$

Insertion Sort

- The outer for loops takes n-1 iterations
- The inner while loop (worst case) runs for n iterations
    - Complexity of insertion sort to be n^2
    - Merge sort is better

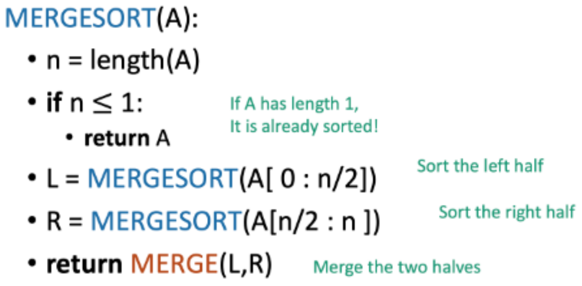


```
def InsertionSort(A):
   for i in range(1,len(A)):
        current = A[i]
        j = i-1
        while j >= 0 and A[j] > current:
            A[j+1] = A[j]
            j -= 1
        A[j+1] = current
```

Insertion Sort

Merge Sort

- Divide and conquer approach
- Recursive approach and divides each problem into two equal sized halves
- Iteration of merge sort divides the problem into half, the length of divisions = log(n)
    - Complexity O(n log n)



```
MERGESORT(A):
  • n = length(A)
  • if n ≤ 1:                 If A has length 1,
                              It is already sorted!
      • return A
  • L = MERGESORT(A[ 0 : n/2])      Sort the left half
  • R = MERGESORT(A[n/2 : n ])      Sort the right half
  • return MERGE(L,R)          Merge the two halves
```

Divide and conquer creates unbalanced division of work

- Quick sort: selection pivot controls size of two blocks
- Maximum-subarray problem
- Square matrix multiplication problem

Division resulting in three + sub problems

- Assuming log has base 2
- If more than two subproblems consider log(k), k is number of sub problems.
- Systematic way of finding these complexities

Recurrence

- Is an equation or inequality that describes a function in terms of its value on smaller inputs.
- Solution $T(n) = O(n\log(n))$

$$T(n) = \begin{cases} O(1), & \text{if } n = 1 \\ 2T(n/2) + O(n), & \text{if } n > 1 \end{cases}$$

- Recurrence says when the **problem size is of size 1,** it is in constant time.
- If $n > 1$, algorithm breaks the input into sub-problems each operation on n/2 size of the data.
- The sub-problem spends $O(n)$ time other than just breaking the bigger problem into sub-problems.

- A recursive algorithm might divide subproblems into unequal sizes, such as 2/3 to 1/3 split. If the divide and combine streps take linear time.
    - This would rise the recurrence $T(n) = T(2n/3) + T(n/3) + O(n)$

Subproblems <u>are not necessarily constrained</u> to being a constant fraction of the original problem size.

- A recursive version of linear search would create just one subproblem containing only one element fewer than the original problem
- Would take constant time **plus the time for the recursive calls it makes** $\mathrm{T}(n) = T(n-1) + O(1)$

- Merge sort on n elements when n is odd, we end up with subproblems of size *floor* $\left(\frac{n}{2}\right)$ and *ceil* $\left(\frac{n}{2}\right)$
    - Neither size is actually $\left(\frac{n}{2}\right)$, because $\left(\frac{n}{2}\right)$ **is not an integer when n is odd!**

$$T(n) = \begin{cases} O(1), & \text{if } n = 1 \\ \lfloor T(n/2) \rfloor + \lceil T(n/2) \rceil + O(n), & \text{if } n > 1 \end{cases}$$

Boundary conditions

- Running time of an algorithm (constant-sized input is a constant) $\mathrm{T}(n) = TO(1)$ for small n. ***State or solve recurrences omit floors, ceilings, and boundary conditions**

Recurrence Relations

- A formula for $T(n)$ where $T(less\ than\ n)$
- Given a recurrence relation for $T(n)$, find a closed form expression for $T(n)$
- Think of $T(1)$ in terms of complexity, $T(1) = O(1)$, constant etc
    - Base Case, not as important when it is constant.
    - Result: $T(n) = O(nlog(n))$

Merge sort

Formulate a recurrence relation of the merge sort:

1. If we have solved the two subproblems of size $\left(\frac{n}{2}\right)$, it takes us $11n$ operations to find result.
    - The use of 11 is arbitrary, it could 10 or 5, order is O(n).
- $T(n)$ where $T(less\ than\ n)$

2. Find a closed form expression for $\mathrm{T}(n)$, that is $T(n) = O(n\log(n))$

The Substitution Method

- A method guaranteed to find right result.
    1. General Substitution Approach
    2. Substitution based on Induction

General substitution approach

- Recurrence Merge Sort:

$$T(n) = 2\,T\left(\frac{n}{2}\right) + \boldsymbol{n}$$

or

$$T(n) = 2\,T\left(\frac{n}{2}\right) + \boldsymbol{O(n)}$$

$$T(n) = 2T(n/2) + 11n$$

- Find is $T(n) = O(n\log(n))$
- Substitutes values of decreasing n into the recurrence equation (get rid of 11)!

The value of n decreaments by half

$$16 \to 8, \qquad 8 \ \to 4, \qquad 4 \ \to 2, \qquad 2 \ \to 1$$

- Replace with $\left(\frac{n}{2}\right)$,

$$T(n) = T(n/2) + c$$

$$T(n/2) = T(n/4) + c$$

$$T(n/4) = T(n/8) + c$$

- Replace with $T\left(\frac{n}{2}\right)$,

$$T(n) = T(n/4) + 2c$$

- Replace with $T\left(\frac{n}{4}\right)$,

$$T(n) = T(n/8) + 3c$$

- Rewrite,

$$T(n) = T(n/2^3) + 3c$$

Pattern is found,

$$T(n) = T(n/2^k) + kc$$

Base condition where $T(n) = 1$

$$T(n/2^k) = T(1)$$

Solve for the value of K,

$$\begin{aligned} \frac{n}{2^k} &= 1 \\ n &= 2^k \\ \log n &= k\log(2) \\ \log(n) &= k \end{aligned}$$

Solve recurrence is $T(n) = T\left(\frac{n}{2}\right) + c$,

- complexity of is $O(\log(n))$

$$\begin{aligned} T(n) &= T(1) + kc \\ &= c \times \log(n) \\ &= \log(n) \\ &\equiv O(\log(n)) \end{aligned}$$

Substitution based on induction

1. Maintain a loop invariant – true at every iteration
2. Proceed by induction

Four steps in the proof by induction:

1. Inductive Hypothesis: The loop invariant holds after the i^{th} iteration
2. Base case: The loop invariant holds before the 1st iteration
3. Inductive step: If the loop invariant holds after the i^{th} iteration, then it holds after the $(i+1)^{st}$ iteration
4. Conclusion: if the loop invariant holds after the last iteration, then we win!

- Solve merge sort recurrence formulation via substitution methods.
- Formulate recurrence,
-

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

Extend above recurrence,

$$\begin{aligned} T(n/2) &= 2T(n/4) + n/2 \\ T(n/4) &= 2T(n/8) + n/4 \end{aligned}$$

Substitute above values in recurrence to find pattern,

$$\begin{aligned} T(n) &= 2\left(2T(n/4) + n\right) + n \\ &= 4T(n/4) + 2n \end{aligned}$$

Thus,

$$\begin{aligned} T(n) &= 2\left(2T(n/4) + n\right) + n \\ &= 4T(n/4) + 2n \\ &= 4\left(2T(n/8) + n/4\right) \\ &= 8T(n/8) + 3n \end{aligned}$$

Pattern,

$$T(n) = 2^k T(\frac{n}{2^k}) + kn$$

Base condition where $T(n) = 1$

- Solve recurrence $\frac{n}{2^k} = 1$
- $k = \log{(n)}$
- Substitute the value of k into equation and back substitute the values in original recurrence formulation

$$\begin{aligned} T(n) &= 2^{\log(n)} T(\frac{n}{2^{\log(n)}}) + \log(n).n \\ &= n.1 + n.\log(n) \\ &= n + n\log(n) \\ &\equiv O(n + n\log(n)) \\ &= O(n\log(n)) \end{aligned}$$

Linear Recurrence

- n decreases from n to $n - 1$
- Solve recurrence using substitution method
- Expand term $T(n)$

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

→

$$T(n-1) = T(n-2) + \log(n-1)$$

$$T(n-2) = T(n-3) + \log(n-2)$$

- Replace $T(n-1) = T(n)$
- Replace $T(n-2) = T(n-1)$

$$T(n) = T(n-2) + \log(n-1) + \log(n)$$

$$T(n) = T(n-3) + \log(n-2) + \log(n-1) + \log(n)$$

- Find Pattern,
- Rewrite form of K,

$$T(n) = T(n-k) + \log(n-(k-1)) + \log(n-(k-2)) + \ldots + \log(n-1) + \log(n)$$

- Set $n - k = 1$,
- Solve for the value of k,

- Thus $k = n$ where $T(n-k) = T(1)$

$$\begin{aligned} T(n) &= T(1) + \log(k-(k-1)) + \log(k-(k-2)) + \ldots + \log(k-1) + \log(k) \\ &= T(1) + \log(1) + \log(2) + \ldots + \log(k) \\ &= 1 + \log(1.2.3. \ldots . k) \\ &= 1 + \log(k!) \\ &= 1 + \log(k^k) \\ &= 1 + k\log(k) \\ &= 1 + n\log(n) \\ &\equiv O(n\log(n)) \end{aligned}$$

Unbalanced Recurrences

- Can use computing complexity via induction BUT
- **<u>Akra-Bazzi is GUARANTEED to work</u>** in a multitude of problems

$$T(n) = T(\lceil n/5 \rceil) + T(7n/10 + 6) + O(n)$$

- <u>Solving via substitution becomes complex</u>
- **<u>Use Akra-Bazzi method</u>**

$$\sum_{i=1}^{k} a_i T(b_i n + h_i(n)) + g(n)$$

- Two subproblem results in, $k = 2$
- $a_1 = 1$
- $b_1(n) = \frac{1}{5}$

- $a_2 = 1$
- $b_2(n) = \frac{7}{10}$
- $g(n) = O(n)$

- Solve to find value of p,

$$\sum_{i=1}^{k} a_i b_i^p = 1$$

Thus, $p = 0.81$

$$1 \times (1/5)^p + 1 \times (7/10)^p = 1$$

Solve,

$$T(n) = O(n^p \left(1 + \int_1^n \frac{g(x)}{x^{p+1}} dx\right))$$

- Since $g(n) = O(n)$
    - $g(x) = O(x)$

$$T(n) = O(n^{0.81} \left(1 + \int_1^n \frac{x}{x^{1.81}} dx\right))$$

Thus,

$$T(n) = O(n^{0.81} \left(1 + \int_1^n x^{-0.81} dx\right))$$

Integrate,

$$\begin{aligned} T(n) &= O(n^{0.81} \left(1 + \left[\frac{x^{0.19}}{0.19}\right]_1^n\right)) \\ &= O(n^{0.81} \left(1 + \left[\frac{n^{0.19}}{0.19} - \frac{1^{0.19}}{0.19}\right]\right)) \\ &= O(n^{0.81} \left(1 + \frac{1}{0.19}\left[n^{0.19} - 1\right]\right)) \\ &\equiv O(n) \end{aligned}$$

Substitution based on Induction

Solve following recurrence,

$$T(n) = T(\lceil n/5 \rceil) + T(7n/10 + 6) + O(n)$$

- Substitution method for solving recurrence comprises two steps,

1. Guess the form of the solution. Requires substantial experience (guess the form of the answer to apply it, use some heuristics) or you can use recursion trees to generate good guesses
2. Use mathematical induction to find the constants and show that the solution works.

* c = constant

- Guess: Solution = $O(n\log(n))$
    - ○ $T(n) = O(n\log(n))$
- Prove $T(n) <= c\, n \log n$, where $c > 0$

  * Holds true to for all positive $m < n$, where $m = \frac{n}{2}$

  - Thus, $T\left(\frac{n}{2}\right) <= C\left(\frac{n}{2}\right)\log(\frac{n}{2})$

$$\begin{aligned} T(n) &\le 2(c(n/2)\log(n/2)) + n \\ &\le cn\log(n/2) + n \\ &= cn\log n - cn\log 2 + n \\ &= cn\log n - cn + n \\ &\le cn\log n. \end{aligned}$$

Recursive form,

$$T(n) = T(n/2) + T(n/2) + 1$$

Solve following recurrence,

- Guess: Solution = T(n) = $O(n)$
- Prove $T(n) <= c\, n$

$$\begin{aligned} T(n) &\le c(n/2) + c(n/2) + 1 \\ &= cn + 1 \end{aligned}$$

- We are off by constant 1 (a lower order term)
- Subtract a lower-order term from previous guess
- Prove $T(n) <= c\, n - d$, where $d >= 0$

Thus $d >= 1$

$$\begin{aligned} T(n) &\le c(n/2) - d + c(n/2) - d + 1 \\ &= cn - 2d + 1 \\ &\le cn - d \end{aligned}$$

Solve following recurrence,

- Pick a constant where function described by $O(n)$, non-recursive component of running time is bounded above by the term $a \cdot n$ for all $n > 0$

Thus,

- $c \cdot n$, where $T(n) <= cn$
- If $\left(-\frac{cn}{10} + 7c + an\right) <= 0$
    - Complexity is $T(n) = O(n)$

$$\begin{aligned} T(n) &\leq c(\lceil n/5 \rceil) + c(7n/10 + 6) + an \\ &\leq cn/5 + c + 7n/10 + 6c + an \\ &= 9cn/10 + 7c + an \\ &= cn + (-cn/10 + 7c + an) \end{aligned}$$

The Master Theorem

- Sub class problems use *The Master Theorem* method.
- Solves recurrence when all of the subproblem are of equal sizes.

Recurrence form,

$$T(n) = aT(\frac{n}{b}) + O(n^d)$$

Solve as,

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

1. a: number of subproblems
2. b: factor by which input size shrinks
3. d: need to do n^d work to create all subproblems and combine solutions

Solve recurrence form,

$$T(n) = 8T(n/2) + n^2$$

1. $a = 8$
2. $b = 2$
3. $\boldsymbol{d = 2}$

- Since $8 > 2^3$
    - **Selection case 3.** $\boldsymbol{d = 2}$

Thus, $T(n) = O(n^3)$

- $log_2 8 = 3$

However, d can be hard to formalise, an alternative formulation can be used,

$$T(n) = aT(\frac{n}{b}) + f(n)$$

1. $a >= 1$
2. $b > 1$

*We have 'a' subproblem, each is of equal size and operates on $\frac{n}{b}$ size of input data

***There is no d**

Solution complexity,

$$T(n) = n^{\log_b a}[U(n)]$$

- Value of $U(n)$ depends on $h(n)$, where $h(n)$,

$$h(n) = \frac{f(n)}{n^{\log_b a}}$$

Relationship between $h(n)$ and $U(n)$,

$$U(n) = \begin{cases} O(n^r), & \text{if } h(n) \text{ is } n^r, r > 0 \\ O(1), & \text{if } h(n) \text{ is } n^r, r < 0 \\ \frac{(\log_2 n)^{i+1}}{i+1} & \text{if } h(n) \text{ is } (\log n)^i, i \geq 0 \end{cases}$$

Problem 1

- Recurrence using The Master Method,

$$T(n) = 8T(n/2) + n^2$$

1. $a = 8$
2. $b = 2$
3. $\boldsymbol{f(n)} = n^2$

Solution,

$$\begin{aligned} T(n) &= n^{\log_b a} U(n) \\ &= n^{\log_2 8} U(n) \\ &= n^3 U(n) \end{aligned}$$

Solve $h(n)$,

$$\begin{aligned} h(n) &= \frac{f(n)}{n^{\log_b a}} \\ &= \frac{n^2}{n^3} \\ &= \frac{1}{n} \\ &= n^{-1} \end{aligned}$$

$h(n) = n^{-1}$

- Solve $U(n)$,

1. **a: number of subproblems**
2. b: factor by which input size shrinks
3. d: need to do n^d work to create all subproblems and combine solutions

$$U(n) = \begin{cases} O(n^r), & \text{if } h(n) \text{ is } n^r, r > 0 \\ O(1), & \text{if } h(n) \text{ is } n^r, r < 0 \\ \frac{(\log_2 n)^{i+1}}{i+1} & \text{if } h(n) \text{ is } (\log n)^i, i \geq 0 \end{cases}$$

- ○ $r = -1$
- ○ Use Case 1

- Recurrence associated with binary search problem!

The Recursion Tree Method

- Most popular application, use before The Substitution Method or The Master Theorem!
- Each node represents the cost of a single subproblem (within the set of recursive function invocations)
- Sum the costs within each level of tree to obtain a set of per-level costs
- Sum all the per-level costs to determine the total costs of all levels of the recursion

"A recursion tree is best used to generate a good guess, which you can then verify by the substitution method. When using a recursion tree to generate a good guess, you can often tolerate a small amount of "sloppiness," since you will be verifying your guess later on. If you are very careful when drawing out a recursion tree and summing the costs, however, you can use a recursion tree as a direct proof of a solution to a recurrence." [4]

Solve following recurrence,

$$T(n) = 3T\lfloor n/4 \rfloor + O(n^2)$$

- Find the upper bound for the solution
- Remember: *floors* and *ceilings* do not matter when solving recurrences
- Create a recursion tree for the recurrence,

$$T(n) = 3T\left(\frac{n}{4}\right) + O(n^2)$$

$$or$$

$$T(n) = 3T\left(\frac{n}{4}\right) + cn^2$$

- Where $c > 0$

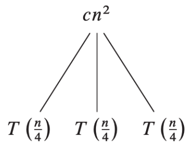
* $c =$ constant

*Assume $n =$ power of 4

This is to account for sloppiness so all subproblems sizes are integers

$$T(n) = 3T(n/4) + cn^2$$

- 3 subproblems, each subproblem has input / 4
- Each iteration n^2



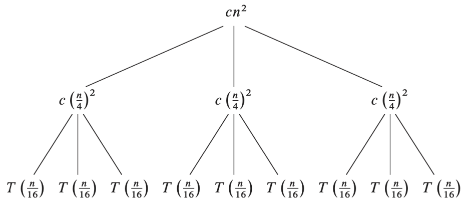
cn^2
$T\left(\frac{n}{4}\right)$
$T\left(\frac{n}{4}\right)$
$T\left(\frac{n}{4}\right)$

- $T\left(\frac{n}{4}\right) = c\left(\frac{n}{4}\right)^2$

Second step,

- $c\left(\frac{n}{4}\right)^2 + c\left(\frac{n}{4}\right)^2 + c\left(\frac{n}{4}\right)^2 = 3c\left(\frac{n}{4}\right)^2 = \frac{3}{16}cn^2$

Expand tree,

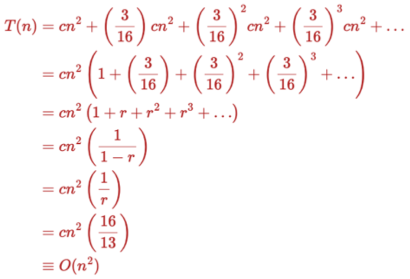


cn^2
$c\left(\frac{n}{4}\right)^2$
$c\left(\frac{n}{4}\right)^2$
$c\left(\frac{n}{4}\right)^2$
$T\left(\frac{n}{16}\right)$
$T\left(\frac{n}{16}\right)$
$T\left(\frac{n}{16}\right)$
$T\left(\frac{n}{16}\right)$
$T\left(\frac{n}{16}\right)$
$T\left(\frac{n}{16}\right)$
$T\left(\frac{n}{16}\right)$
$T\left(\frac{n}{16}\right)$
$T\left(\frac{n}{16}\right)$

Problem size is $\frac{n}{16}$, there are 9 sub problems

- $T\left(\frac{n}{16}\right) = c\left(\frac{n}{16}\right)^2$
- $9\,c\left(\frac{n}{16}\right)^2 = \left(\frac{3}{16}\right)^2 cn^2$

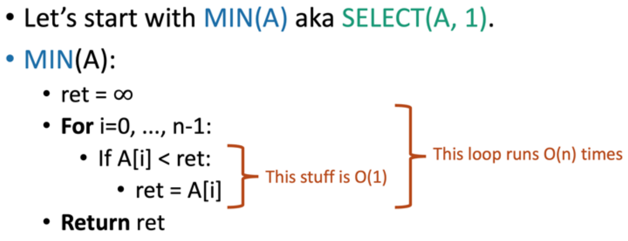
Find pattern,



$$
\begin{aligned}
T(n) &= cn^2 + \left(\frac{3}{16}\right)cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \left(\frac{3}{16}\right)^3 cn^2 + \ldots \\
&= cn^2\left(1 + \left(\frac{3}{16}\right) + \left(\frac{3}{16}\right)^2 + \left(\frac{3}{16}\right)^3 + \ldots\right) \\
&= cn^2\left(1 + r + r^2 + r^3 + \ldots\right) \\
&= cn^2\left(\frac{1}{1-r}\right) \\
&= cn^2\left(\frac{1}{r}\right) \\
&= cn^2\left(\frac{16}{13}\right) \\
&\equiv O(n^2)
\end{aligned}
$$

The Select(A, k) Problem

- Selects the k smallest element in a list A
- A is a list of distinct elements
- If $k = 0$, returns smallest element in list
- If $k = n = len(A)$, returns the maximum element
- $Select(A, n/2)$, returns the median element



- Let's start with MIN(A) aka SELECT(A, 1).
- MIN(A):

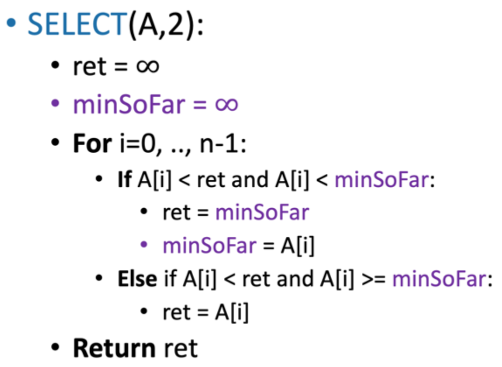
```
• ret = ∞
• For i=0, ..., n-1:
    • If A[i] < ret:
        • ret = A[i]
• Return ret
```

This stuff is O(1)

This loop runs O(n) times

If you cannot sort (list is large and distributed across multiple machines), write the function by keeping track of minimum element, then traverse through list.

- Complexity $O(n)$



```
• SELECT(A,2):
    • ret = ∞
    • minSoFar = ∞
    • For i=0, .., n-1:
        • 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
```

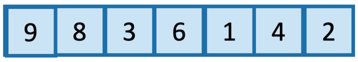
- Find minimum element
- Keep track of two elements, minimum and second minimum

- Find median element
- Keep track of $\frac{n}{2}$ set of minimum element in list
- Complexity $O(n^2)$
    - Merge sort has complexity $O(n \log(n))$, which is better complexity

Rely on Divide and Conquer (recursive formulation) resulting in recurrences.

Divide and Conquer Based Select(A, K)

- Quick Sort, pivot randomly to divide lists into two parts



9	8	3	6	1	4	2

- Pivot = 6
- element < 6, left sub-list (L)
- elements > 6, right sub-list (R)

3	1	4	2

L = array with things smaller than A[pivot]

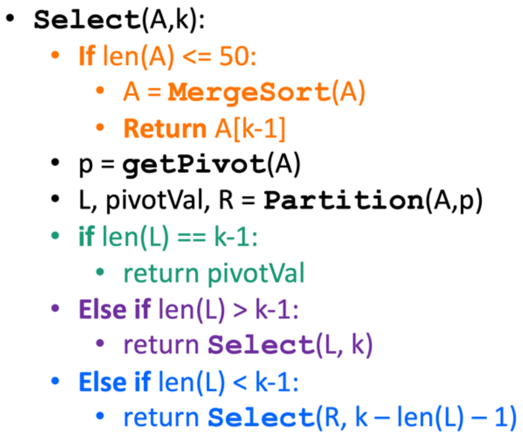
9	8

R = array with things larger than A[pivot]

- If $k = 5$,
- Pivot guess is lucky
- List is perfectly split!
- Thus, all elements < k are in left array
    - Return pivot based on check $(if(len(L) = k - 1))$

BUT if…

- If $len(L) > k - 1$,
    - k-th smallest element is in **left** list, apply $\boldsymbol{select(L, k)}$
- If $len(L) < k - 1$,
    - k-th smallest element is in **right** list, apply $\boldsymbol{select(R, k - len(L) - 1}$



```
• Select(A,k):
    • If len(A) <= 50:
        • A = MergeSort(A)
        • Return A[k-1]
    • p = getPivot(A)
    • L, pivotVal, R = Partition(A,p)
    • if len(L) == k-1:
        • return pivotVal
    • Else if len(L) > k-1:
        • return Select(L, k)
    • Else if len(L) < k-1:
        • return Select(R, k – len(L) – 1)
```

Algorithm,

$$T(n) = \begin{cases} T(len(L)) + O(n), & \text{if } len(L) > k - 1 \\ T(len(R)) + O(n), & \text{if } len(L) < k - 1 \\ O(n), & \text{if } len(L) = k - 1 \end{cases}$$

Apply The Master Theorem

- **You can't as it dictates that your subproblem must be of equal sizes**
- Complexity would be T$(n) = O(n)$, linear

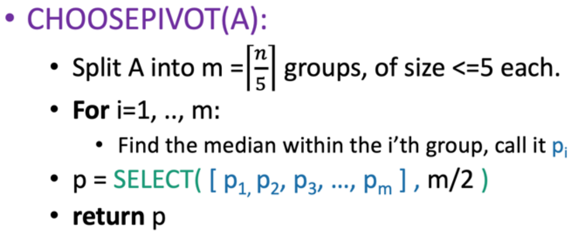
$$a = 1, b = 2, d = 1$$
$$a = 1, b = \frac{10}{7}, d = 1$$

$3n/10 < len(L) < 7n/10$
$3n/10 < len(R) < 7n/10$

- If you applied The Master Theorem you can't guarantee pivot will not breast worse than 30% or 70%
- Find guaranteed pivot?

Apply Median of Median Method for Pivot Selection

- Median breaks list into two subproblems
- Complexity $O(n)$, linear
- Break list, each sub-list has 5 elements
- Compute medium of each list, compute median of the medians
- Median of the median used as the pivot



- CHOOSEPIVOT(A):
  - Split A into m $= \left\lceil \frac{n}{5} \right\rceil$ groups, of size <=5 each.
  - **For** i=1, .., m:
    - Find the median within the i'th group, call it p_i
  - p = SELECT([$p_1, p_2, p_3, \ldots, p_m$] , m/2)
  - **return** p

- Choose Pivot,

$$3n/10 < len(L) < 7n/10$$
$$3n/10 < len(R) < 7n/10$$

- Cannot use recurrence, divide, and conquer strategy instead

$$T(n) = T(7n/10) + O(n)$$

- Modify recurrence,

$$T(n) = T(n/5) + T(7n/10) + O(n)$$

Solution, solve using Akra-Baazi method or using induction

- Resulting in linear time complexity algorithm

Activity 1

- **(1)** Solve following recurrence using substitution method:

a. $T(n) = \begin{cases} nT(n-1), & \text{if } n > 1 \\ 1, & \text{if } n = 1 \end{cases}$

b. $T(n) = \begin{cases} T(n-1) + \log n, & \text{if } n > 1 \\ 1, & \text{if } n = 1 \end{cases}$

c. $T(n) = \begin{cases} 3T(n/2) + n, & \text{if } n > 1 \\ 1, & \text{if } n = 1 \end{cases}$

1. Recursive relations normally reflect the runtime of recursive algorithms.

Substitution method for solving recurrence comprises two steps,

- Guess the form of the solution. Requires substantial experience (guess the form of the answer to apply it, use some heuristics) or you can use recursion trees to generate good guesses
- Use mathematical induction to find the constants and show that the solution works.

a. $T(n) = \left\{\begin{matrix} nT(n-1), & if\ n > 1 \\ 1, & if\ n = 1 \end{matrix}\right\}$ ← Given base case

Base case: $T(1) = 1$

Recursive case: where $n > 1$,

$$T(n) = n \cdot T(n-1)$$

Expand $T(n-1)$ using same relation,

$$T(n-1) = (n-1) \cdot T(n-2)$$

Expand until base case $T(1)$ is found,

$$T(n) = n \ \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 2 \ \cdot 1 \cdot T(1)$$

Substitute the base case,

Since $T(1) = 1$, this value can be used

$$T(n) = n \ \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 2 \ \cdot 1 \cdot 1$$

Simplify,

$$T(n) = n \ \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 2 \ \cdot 1$$

Closed form solution $T(n)$, factorial is the product of all positive integers from 1 to n, which is $n!$

$$T(n) = \left\{\begin{matrix} nT(n-1), if\ n > 1 \\ 1, \qquad\quad if\ n = 1 \end{matrix}\right\}$$

$$T(n) = n!$$

Time complexity,

$$O(n!)$$

The function is the factorial of n, where $T(n)$ grows at a rate (very fast) that is proportional to n! Additionally this exponential growth is not practical for large values.

---

b. $T(n) = \left\{\begin{matrix} T(n-1) + \log n\,, & if\ n > 1 \\ 1, & if\ n = 1 \end{matrix}\right\}$

Base case: $T(1) = 1$

Recursive case: where $n > 1$,

$$T(n) = T(n-1) + \log\,(n)$$

Expand $T(n-1)$ using same relation,

$$T(n-1) = T(n-2) + \log\,(n-1)$$

Expand until base case $T(1)$ is found,

$$\begin{aligned} T(n) &= T(n-1) + \log(n) \\ &= (T(n-2) + \log(n-1) + \log(n) \\ &= T(n-3) + \log(n-2) + \log(n-1)) + \log(n) = \ \dots \end{aligned}$$

Continued,

$$T(n) = T(1) + \log(2) + \log(3) + \ \dots + \log\,(n)$$

Simplify,

$$T(n) = 1 + \log(2) + \log(3) + \ \dots + \log\,(n)$$

Sum logs,

Logarithmic rule: $\log(a) + \log(b) = \log\,(a \cdot b)$,

$$T(n) = \log(2 \cdot 3\ \cdot \dots \cdot n) + 1$$

$log(2 \cdot 3\ \cdot \dots \cdot n)$ is factorial

Simplify,

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

$$T(n) = \begin{cases} T(n-1) + \log n\,, & if\ n > 1 \\ 1, & if\ n = 1 \end{cases}$$

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

Time complexity,

$$O(\log(n!) + 1) = \ O(n \log n)$$

---

c. $T(n) = \begin{cases} 3T(n/2)\ +\ n, & if\ n > 1 \\ 1, & if\ n = 1 \end{cases}$

Base case: $T(1) = 1$

Recursive case: where $n > 1$,

$$T(n) = 3T(n/2) + n$$

Expand $T(n/2)$ using same relation,

$$T(n/2) = 3T\left(\frac{n/2}{2}\right) + n/2$$

$$T(n/2) = 3T(n/4) + n/2$$

(3)

Expand until base case $T(1)$ is found,

$$\begin{aligned} T(n) &= 3T(n/2) + n \\ &= 3[3T(n/4) + n/2]\ +\ n \end{aligned}$$

(3^2)

$$= 9T(n/4) + 3\,n/2\ +\ n$$

Substitute,

$$T(n/4) = 3T(\frac{n/4}{2})\ + n/4$$ ←

Substitute for recursive implementation,

$$T(n/2) = 9[3T(n/8) + n/4\ +\ 3\,n/2\ + n$$

(3^3)

$$= 27T(n/8)\ + 9n/4\ + 3\ (n/2)\ + n$$

Substitute,

$$T(n/8) = 3T(\frac{n/8}{2})\ + n/8$$ ←

Substitute,

$$T(n/8) = 27(\ 3T(\frac{n/8}{2})\ + n/8)\ + (9/4)\,n + (3/2)\,n\ +\ n$$

(3^4)

$$T(n/8) = 81T(n/16)\ + 27\,n/8)\ + (9/4)\,n + (3/2)\,n\ +\ n$$ ←

Find Pattern,
$3, 3^2, 3^3, 3^4 \ldots 3^k$

Simplify using k implementations,

$$= 3^k T\left(\frac{n}{2^k}\right) + n\ [\frac{3}{2} + \left(\frac{3}{2}\right)^2 + \left(\frac{3}{2}\right)^3 + \left(\frac{3}{2}\right)^4 + \ \ldots + \left(\frac{3}{2}\right)^{k-1}\ ]$$

Simplify summation,

$$= 3^k T\left(\frac{n}{2^k}\right) + n\ [\,\Sigma\left(\frac{3}{2}\right)^k\ ]$$

Simplify apply $\log_2$ of both sides,
Exponential form,

$$2^k\ = n$$
$$log_{2^k} = log_n$$
$$k\ = log_n$$

$$= 3^{log_n}\ T\left(\frac{n}{n}\right) + n[\frac{3^k}{2^k}]$$

Substitute,

$$= 3^{log_n}\ T(1) + n[\frac{3^{\log_2 n}}{2^{\log_2 n}}]$$

Logarithmic rule: $a^{\log_b c}\ = b^{\log_a c}$

Simplify,

$$= n^{\log_{3_2}} + n\,[\frac{n^{\log_2 3}}{n^{\log_2 2}}]$$

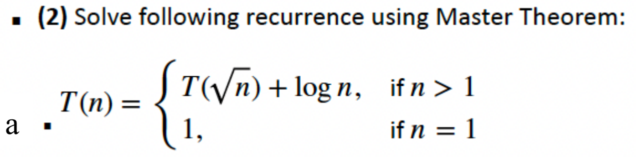
$$= n^{\log_2 3} + n\,[\frac{n^{\log_2 3}}{n}]$$

$$= 2n^{\log_2 3}$$

Time complexity,

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

Activity 2



- **(2)** Solve following recurrence using Master Theorem:

a ▪ $T(n) = \begin{cases} T(\sqrt{n}) + \log n, & \text{if } n > 1 \\ 1, & \text{if } n = 1 \end{cases}$

- The Master Theorem is a formula that solves recurrences when all the subproblems are the same size!
- There are two forms of The Master Theorem
  - 1. Where the equation suits the recurrence of the form
  - 2. Alternative Formulation is needed as the equation does not suit the recurrence of the form
- The recurrence requires three parameters (options),
- a: number of subproblems, b: factor by which input size shrinks, d: need to do n^dwork to create all the subproblems and combine solutions.

a. $T(n) = \begin{Bmatrix} T(\sqrt{n}) + \log n\,, & if\ n > 1 \\ 1, & if\ n = 1 \end{Bmatrix}$

$O(n\log(n))$

Equation does not suit the recurrence of the form, hence using formulation (2) Alternative formulation is used

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

Rewrite the recurrence,

$$T(n) = T(\sqrt{n}) + \log(n)$$

$$= T(n^{1/2}\ ) + \log(n)$$

As per textbook example [4],
Similar to recurrence relation $S(m) = S(m/2)\ + m$
Where, $S(m)$ denotes time complexity of $T(2^m)$

Substitute $n = 2m$,
$n^{1/2} = 2^{m/2}$
$m = \log{(m)}$

$$T(2^m) = T\left(2^{m/2}\right) + \log{(2m)}$$

Simplify,

$$\log{(2\mathrm{m})} = m \log_2 2 = m = 1$$

$$T(2^m) = T\left(2^{m/2}\right) + m$$

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

$$S(m) = S\left(\frac{m}{2}\right) + m\log(m)$$

Coefficient of T in the recurrence,
a: 1

Factor by which n is divided in the recurrence/ problem size,
b: 2

Asymptotically positive for all sufficiency large n/ cost of outside the recursive call,
d: $f(n) = 1$

Solution as per Master method,

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

$$if\ a = b^d$$
$$1 \neq 2^1$$

Since $a = 1$ and $b = 2$, $a\ \neq b^d$ for any non-negative integer d
Case 2, where $d = 0$, as $\log n$ does not use n,

$$if\ a < b^d$$
$$1 < 2^1$$

Solution as per the Master Theorem,

$$n = 2^m$$
$$T(2^m) = T(2^{(m/2)})\ +\ m$$
$$T(n) =\ O\left(\mathrm{n}^{\mathrm{d}}\right) = O(1)$$

Thus,

$$S(m) = O(m)$$

For $n = 2^m$,

$$T(n) = S(m) = O(\log(n))$$

Time Complexity in The Master Solution is given,
$T(n) = O(\mathrm{n}^{\mathrm{d}}(\log(n)^k))$
k is a non-negative constant
d is a non-negative integer such that $a = b^d$

$$T(n) = O(\mathrm{m}^1(\log^0 m))$$
$$T(n) = O(\log(\log n))$$

Activity 3

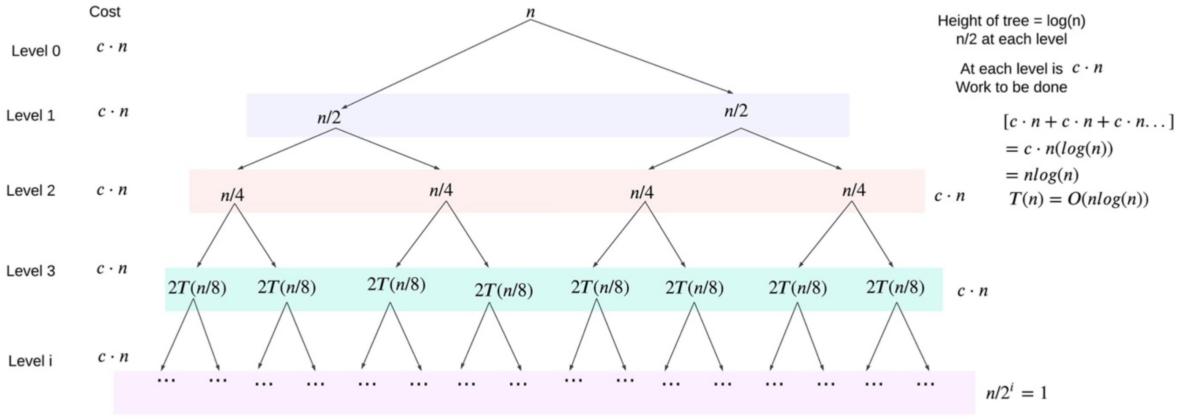
- **(3)** Solve following recurrence using recursion tree method:

a . $T(n) = 2T(n/2) + cn$

Recurrence relation,

$$T(n) = 2T(n/2) + cn$$

Recursion Tree Method, break down recurrence relation into smaller subproblems (tree structure). At each level of tree, the subproblem is broken into smaller subprograms, each of size $n/2$, repeating until base case found $n = 1$.



The height of tree is $log\, n$, dividing $n/2$ at each level, total work done at all levels (cn)

$$T(n) = 2T(n/2) + cn$$

Expand,

$$T(n) = 2[2T(n/4) + c(n/2)] + cn$$
$$= 4T(n/4) + 2cn$$

$$T(n) = 4[2T(n/8) + c(n/4)] + 2cn$$
$$= 8T(n/8) + 3cn$$

Pattern identified!
Find base case where $n/2^i = 1$,

$$T(n) = 2^i\, T(n/2^i) + i \cdot cn$$

$i = \log_2(n)$,

$$T(n) = 2^{\log_2 n}\, T(n/2^{\log_2 n}) + \log_2 n \cdot cn$$

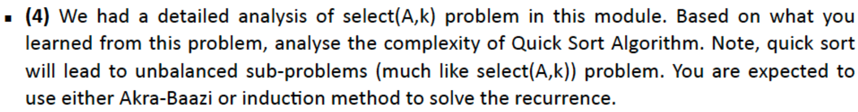
Simplify,

$$T(n) = n \cdot T(1) + c \cdot n \cdot \log_2 n$$

Since $T(1)$ is a constant, total work at each level · total height,
$cn \cdot \log_2 n$

$$T(n) = O(n \log_2(n))$$

Activity 4

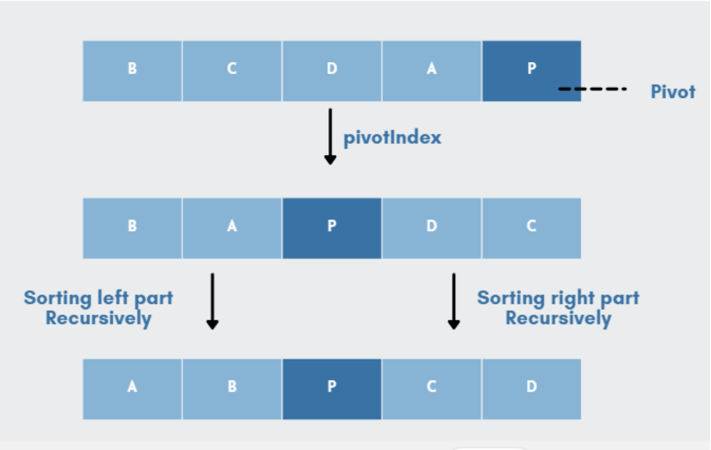


- **(4)** We had a detailed analysis of select(A,k) problem in this module. Based on what you learned from this problem, analyse the complexity of Quick Sort Algorithm. Note, quick sort will lead to unbalanced sub-problems (much like select(A,k)) problem. You are expected to use either Akra-Baazi or induction method to solve the recurrence.

Analyse the complexity of Quick Sort Algorithm,

Quick Sort algorithm based on divide and conquer approach, it is a fast algorithm '…uses $nlog(n)$ comparisions to sort an array of n elements…' [5]. Like Select(A, k) problem from Module 10 [6], Quick sort we can choose a pivot randomly and divide lists into two parts. It splits a large problem into smaller subproblems, best case time complexity $O(nlog(n))$ and space complexity of $O(log(n))$. As Quick Sort has an $O(\log n)$ space complexity it is recommended when space is restricted [5].

Quick Sort Steps,
Using 3 steps, 1. Pick an element, 2. Divide the problem set, move smaller parts to the left of pivot and larger items to right. 3. Repeat steps to combine sub solutions stored.



Quick Sort Algorithm Overview, [5]

- The time complexity of Quick Sort,

$$T(n) = T(k) + T(n-k-1) + O(n)$$

$T(n)$ Time complexity of sorting array of size n
$T(k)$ and $T(n-k-1)$ are the two recursive calls (also known as $(T(L) + T(R))$
$O(n)$ Time partitioning, selecting pivot and rearranging elements

1. **Best Case Complexity,**

- Algorithm finds pivot in the middle element or near middle
- $O(n \log(n))$ time complexity

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

2. **Worst Case Complexity,**

- Algorithm finds pivot selecting largest or smallest element
- Acts similarly to the Select(A, k) problem
- $O(n^2)$ time complexity

$$T(n) = T(0) + T(n-1) + O(n)$$

<u>Quick Sort will lead to unbalanced subproblems</u> if the pivot finds results in one subproblem that is larger than the other.

- The recurrence relation outline for Quick Sort,

$$T(n) = T(L) + T(R) + O(n)$$

$T(n)$ Time complexity of sorting array of size n
$T(L)$ or $T(R)$ Time sorting Left or Right side
$O(n)$ Time partitioning, selecting pivot and rearranging elements

Complexity of Quick Sort algorithm is based on $len(L)$ and $len(R)$

$$T(n) = \begin{Bmatrix} T\big(len(L)\big) + O(n), & if\ len(L) > k-1 \\ T\big(len(R)\big) + O(n), & if\ len(L) < k-1 \\ O(n), & if\ len(L) = k-1 \end{Bmatrix}$$

Master Theorem Method Module 10 Recurrence

- Module 10 [6], used The Master Theorem, even though quick sort random pivot could leave to unequal sizes of lists. The module assumed that the pivot selected equal L and R sizes and resulted in linear time complexity $T(n) = O(n)$.

Assumed pivot selection,

$$3n/10 < len(L) < 7n/10$$
$$3n/10 < len(R) < 7n/10$$

$a: 1, \qquad b = 10/7, \qquad d = 1$

- The recurrence relation,

$$T(n) = T(7n/10) + O(n)$$
$$\equiv O(n^d) = O(n)$$

But we can't guarantee that the pivot selection will always bread '…not worse than 30% and 70%…' [6].

Median of the Median Method Module 10 Recurrence

- To find a median of list without computing the median
- The median breaks the list into two equal subproblems (lists)
- The recurrence **modified and solved using Akra-Baazi method** or using induction
- Resulting in linear time complexity $T(n) = O(n)$

This method of pivot selection guarantees,

$$3n/10 < len(L) < 7n/10$$
$$3n/10 < len(R) < 7n/10$$

Modify recurrence due to divide and conquer strategy,

$$T(n) = T(n/5) + T(7n/10)\, O(n)$$

Akra-Baazi Method/ Theorem Steps Module 10 Recurrence

Akra-Baazi is designed for problems which divide into subproblems with a fixed proportion on their size at each iteration, to analyse asymptotic behaviour [7].

Akra-Baazi Theorem,

$$T(n) = \sum\nolimits_{i=1}^{k} a_i T\big(b_i n + h_i(n)\big) + g(n)$$

Where a_i and b_i are constants such that,

- $a_i > 0$
- $0 < b_i < 1$
- $g(n) \in O(n^c)$
- $h_i(x) \in O(\frac{n}{(logn)^2})$

Modify recurrence from Module 10

$$T(n) = T(n/5) + T(7n/10)\, O(n)$$

Satisfy conditions $a_i > 0$,

$$a_1 = a_2 = 1$$

Satisfy conditions $0 < b_1, b_2 < 1$,

$$b_1 = 3/10$$
$$b_2 = 4/5$$

Where a_i and b_i are constants such that,

$$a_1 = a_2 = 1$$
$$b_1 = 3/10$$
$$b_2 = 4/5$$

Find p such that,

$$\sum_{i=1}^{k} a_i b_i^p = 1$$

Assume p = 1,

$$\sum_{i=1}^{k} a_i b_i^p = a_1 b_1 + a_2 b_2 = b_1 + b_2 = 1$$

Simplified form,

$$= a_1 b_1^{p_1} + a_2 b_2^{p_2} = 1$$

$$2\left(\frac{1}{5}\right)^{p1} + 2\left(\frac{7}{10}\right)^{p2} = 0.4^{p1} + 0.2^{p2} \approx 1$$

Therefore, $p \approx 1$,

Satisfy conditions $g(n) = O(n^c)$ for constant c,
Where $g(n) = O(n^1)$

There is only one level of recursion, for h_1, h_2 such that $h_i(n) = O(\frac{n}{(logn)^2})$

Evaluate Akra-Baazi,

$$T(n) = 0\left(n^p\left(1 + \int_1^n u^{-p} du\right)\right)$$

Substitute value p = 1,

$$\boldsymbol{T(n) = O(n^p logn) = O(n \log n)}$$

Therefore, this recurrence based on Module 10 has worst time complexity of O($nlog(n)$).

Additional Notes:
It does not result in Linear time complexity as the value of p = 1 is an approximate. The recurrence relation for time complexity of Quick Sort as stated above is

$$T(n) = T(k) + T(n-k-1) + O(1)$$

If the pivot point is selected at random than $k = (n-1)/2$ on average therefore it is not possible to find a value of p that would make the time complexity linear, this is the same for induction. However, if we used ‘median of medians’, as shown above from Module 10 to select the pivot element it is possible to obtain a linear time complexity.

Additional Example – Difficult for recursion tree but easy for Akra Baazi

- This example is difficult for recursion trees but easy for Akra-Baazi theorem [7]

$$T(n) = T(3n/4) + T(n/4) + O(n)$$

Where a_i and b_i are constants such that,
$a_i = 1, b_i = 3/4, h_i = 0$
$a_j = 1, b_j = 1/4, h_j = 0$
$g(n) = n$

Find p such that,

$$\sum_{i=1}^{k} a_i b_i^p = 1$$

Example,

$$p = \sum_{i=1}^{k} a_i b_i^p = 1\left(\frac{3}{4}\right)^p + 1\left(\frac{1}{4}\right)^p = 1$$

$$\left(\frac{3}{4}\right)^p + 1\left(\frac{1}{4}\right)^p \approx 1$$

$\approx p = 1$,
Close to zero, use 1 as approximation,

$$\left(\frac{3}{4}\right) + \left(\frac{1}{4}\right) - 1 = 0$$

Substitute values for p,
$\in = 1 - p = 1 - 1 = 0$

$$T(n) = O(n^{1+\in})$$

Function grows asymptotically slower than $n^{1+\in}$ for any positive constant $\in$

Evaluate Akra-Baazi,

$$T(n) = 0\left(n\left(1 + \int_1^n \frac{1}{u} du\right)\right)$$

Substitute values,
Since $\in = 0$,

$$\int_1^n \frac{1}{u} du = \log(n) - \log(1) = \log(n)$$

$$T(n) = O\left(n^{1+0}\log(n)\right)$$

$$= O(n\log(n))$$

Since $\in= 0$, any function that grows asymptotically slower than $nlog(n)$ would also be $O(n\log(n))$.

Bibliography

[1] An insightful Techie, "Data Structures, Big 'O' Notations and Algorithm Complexity," 30 12 2017. [Online]. Available: https://www.youtube.com/watch?v=L7nYZ19zPqw. [Accessed 12 9 2023].

[2] T. a. L. C. a. R. R. a. S. Coreman, "Chapter 4 ," in *C. Divide-and-Conquer*, Cambridge, Massachusetts London, England , The MIT Press , 2022, pp. 76-115.

[3] Code Academy, "Asymptotic Notation," 2023. [Online]. Available: https://www.codecademy.com/learn/cspath-asymptotic-notation/modules/cspath-asymptotic-notation/cheatsheet. [Accessed 12 9 2023].

[4] T. a. L. Coreman, "Divide-and-Conquer, Chapter 4," in *CLRS*, MIT Press Academic, 2022.

[5] P. H, "An Overview of QuickSort Algorithm," Towards Data Science, 11 3 2022. [Online]. Available: https://towardsdatascience.com/an-overview-of-quicksort-algorithm-b9144e314a72. [Accessed 12 9 2023].

[6] Deakin University, " Algorithm Analysis: The Select(A,k) Problem," n.d. [Online]. Available: https://d2l.deakin.edu.au/d2l/le/content/1316240/viewContent/7023569/View. [Accessed 12 9 2023].

[7] J. Erickson, *Solving Recurrences,* Champaign, IL, United States : University of Illinois Urbana-Champaign, 2022.