COMPSCI 2C03
Data Structures & Algorithms

Gulkaran Singh

Fall 2023

Contents

1 Stacks

Stacks are linear data structures that arrange elements in sequential order. Here are some key properties:

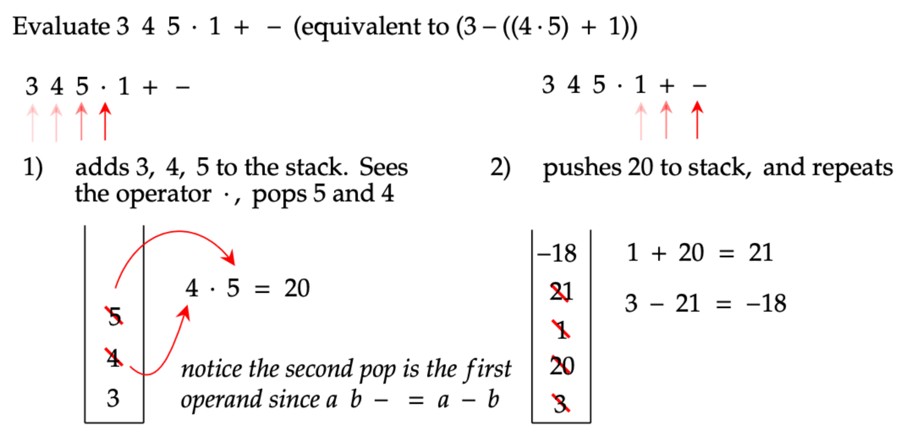
- LIFO – Last In, First Out. This means the last element added to the stack is the **first** out.
- **push()** – adds an item to the TOP of the stack
- **pop()** – removes an item to the TOP of the stack

1.1 Postfix Expression Evaluation

Postfix expressions (e.g. 5 3 + turns into 5 + 3 which is infix) can be computed with stacks. The algorithm has the following steps:

- proceed left to right through the expression
- push operands (values) to the stack
- if an operator, pop twice and apply operator
- push the result back onto the stack

1.1.1 Example


Evaluate 3 4 5 · 1 + − (equivalent to (3 – ((4 · 5) + 1))
3 4 5 · 1 + −
1) adds 3, 4, 5 to the stack. Sees the operator ·, pops 5 and 4
5
4
3
4 · 5 = 20
notice the second pop is the first operand since a b − = a − b
3 4 5 · 1 + −
2) pushes 20 to stack, and repeats
−18
21
1
20
3
1 + 20 = 21
3 − 21 = −18

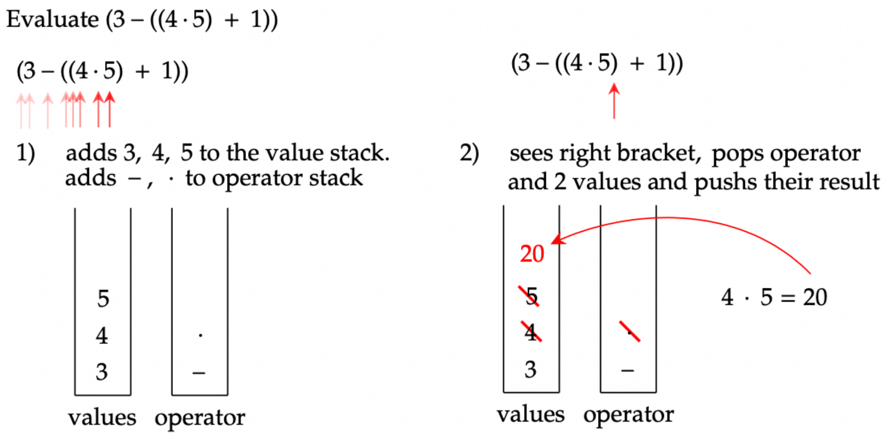
1.2 Dijkstra's Infix Evaluation

This a two-stack approach to solving infix (e.g. 5 + 3) expressions. Note this algorithm requires full and perfect bracketing. Here are the following steps for the algorithm:

- push operands onto value stack

- push operator onto operator stack
- if a right bracket, pop operator and two values
- push the result back into value stack

1.2.1 Example


Evaluate (3 – ((4 · 5) + 1))
(3 – ((4 · 5) + 1))
1) adds 3, 4, 5 to the value stack. adds –, · to operator stack
5
4
3
·
–
values
operator
(3 – ((4 · 5) + 1))
2) sees right bracket, pops operator and 2 values and pushes their result
20
5
4
3
·
–
4 · 5 = 20
values
operator

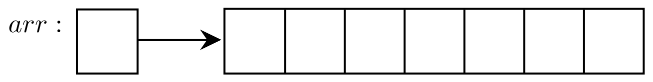
2 Queue

Queues are also linear data structures that arrange elements in sequential order. Here are some key properties:

- FIFO – First In, First Out. The first element in is the first element to leave. Think of queues as lines at the ATM!
- **enqueue()** – add an item to the BACK of the queue
- **dequeue()** – deletes an item at the FRONT of the queue
- elements can only LEAVE the way they entered. There is no variety unlike stacks.

3 Array Implementations

An array is an **indexable** linear data structure that has a **fixed length**. Note the memory is contigious. In Java and C, arrays are auto-initialized to 'null' or zero.



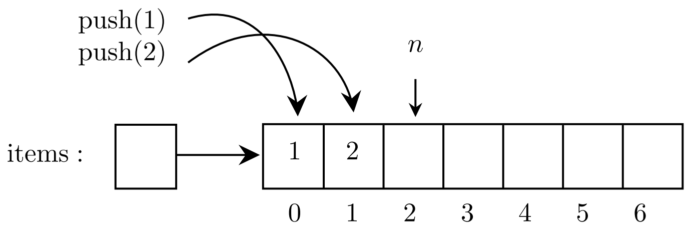
if arr[0] at 0x001, then arr[3] would be at :

$3 \cdot 4$ bytes (if int arr) $= 12$ bytes from arr[0]

3.1 Stack Implementation with Arrays

To implement a stack with a fixed-size array, we need the following:

- an array *items* of size *cap*
- variable n to track the number of items
- **push(item)** – adds item at location n, then n++
- **pop()** – removes item from location $n-1$, then $n--$



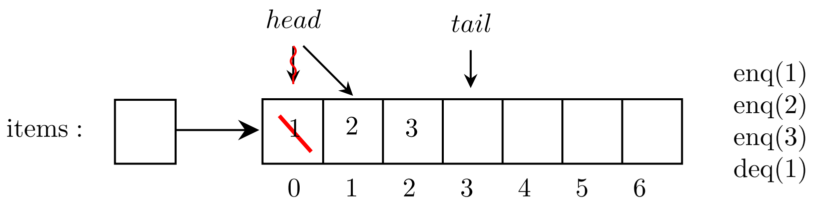
let n= # of items in stack & index of the next available

```
Stack(cap):
  items: array
  n: int <- 0
  size: int <- cap

push(item):
  if stack_full():
    raise error
  items[n] = item
  n++

pop():
  if stack_empty():
    raise error
  n--
  return items[n]
  # notice we don't need to remove items[n] since it will be
  ↪  overwritten if you push another item
```

3.2 Queue Implementation with Arrays

To implement a queue with a fixed-size array, we need the following:

- an array *items* of size *cap*
- variable *head* and *tail* to track the first and last item
- **enq(item)** – adds item at location *tail*, then $tail++$
- **deq()** – removes item at location *head*, then $head++$



This introduces a problem however. What do we do when our *tail* reaches the end of the array? For this, we introduce the 'circular array' implementation. We simply make the *head* and *tail* loop around to the beginning the array by using **modular arithmetic**!

$$head = (head + 1)\ \%\ \text{len}(items)$$
$$tail = (tail + 1)\ \%\ \text{len}(items)$$

```
enq(item):
  if queue_full():
    raise error
  items[tail] = item
  tail = (tail + 1) % len(items)

deq():
if queue_empty():
  raise error
head = (head + 1) % len(items)
```

4 Linked Lists

A linked list is a dynamically sized linear data structure. It includes the following:

- *head* – points to the **first** node in the linked list
- *tail* – points to the **last** node in the linked list
- *node.next* – points to the next node in the linked list

All three are references to Nodes and can be null if neccessary. We typically have two special cases with linked lists:

- Singleton – linked list of length 1 -¿ head == tail
- Empty – head == tail == null

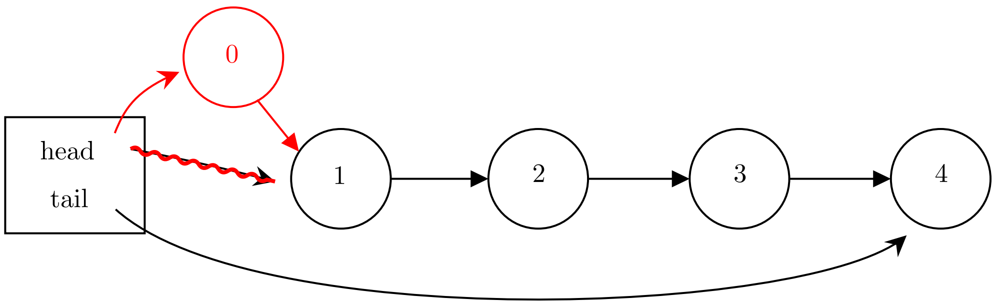
4.1 Basic Operations

Here are the most common examples of algorithms associated with the operations of linked lists.

4.1.1 Prepend

Adding a Node to the head of the linked list.

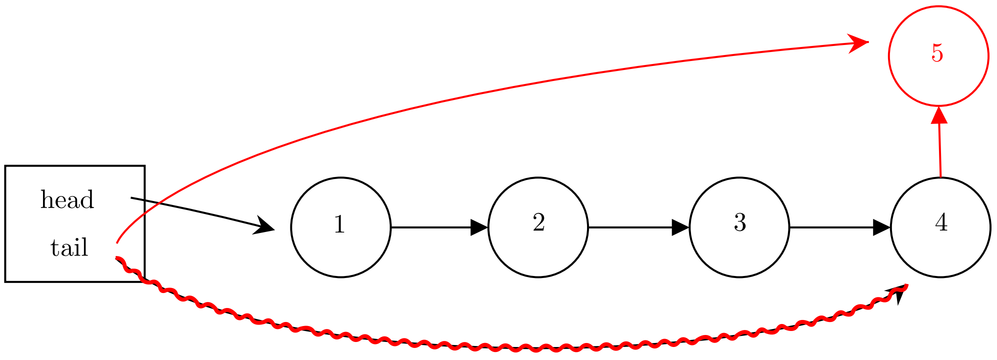
```
n = Node(item)
if isEmpty():     # empty
  head = n
  tail = n
else:             # non empty
  n.next = head
  head = n
```



4.1.2 Append

Adding a Node to the end (tail) of the linked list.

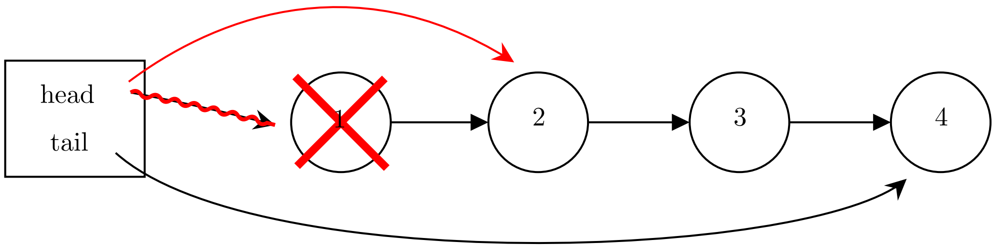
```
n = Node(item)
if isEmpty():     # empty
  head = n
  tail = n
else:             # non empty
  n.tail.next = n
  tail = n
  n.next = null
```



4.1.3 Delete First

Deleting the head of the linked list, returning the first item

```
if head == null:      # empty
  raise error

if head == tail:      # singleton
  item = head.item
  head = null
  tail = null
  return item

item = head.item      # general
head = head.next
return item
```



4.1.4 Search

Tranversing through the linked list to find if the key exists.

```
curr = head
while curr != null:
  if curr.item == key:
    return curr
  curr = curr.next
return null
```

4.2 Stack and Queue Implementations

- **Stack** – use prepend for push(), delete_first() for pop()
- **Queue** – use append for enqueue(), delete_last() for dequeue()

5 Algorithm Analysis

We denote $T(n)$ to be a time complexity function for an algorithm. It considers the number of basic instructions executed (assignments, operations, indexing, function calls, etc.)

$T(n)$ – worst case $\qquad T_{best}(n)$ – best case $\qquad T_{avg}(n)$ – avg case

5.1 Formal Definitions

Big-O: $T(n) \in O(f(n))$ if there exists constants $c > 0, n_0 \geq 0$ such that

$$0 \leq T(n) \leq c \cdot f(n)$$

for all $n \geq n_0$. Best approach is direct, find c, n_0, show $T(n) \geq 0$ and show $T(n) \leq c \cdot f(n)$

Big-Ω: $T(n) \in \Omega(f(n))$ if there exists constants $c > 0, n_0 \geq 0$ such that

$$T(n) \geq c \cdot f(n)$$

for all $n \geq n_0$. This definition is equivalent to

$$T(n) \in \Omega(f(n)) \qquad\qquad T(n) \notin O(f(n)) \qquad\qquad f(n) \in O(T(n))$$

Big-Θ: $T(n) \in \Theta(f(n))$ if and only if

$$T(n) \in O(f(n)) \wedge T(n) \in \Omega(f(n))$$

for all $n \geq n_0$.

6 Working with Big-O

6.1 Complexity Classes

Here are the most common complexities and their names.

Name	Big-O	Example
Constant	$O(1)$	Indexing into an array
Logarithmic	$O(\log n)$	Binary search
Linear	$O(n)$	Searching a linked list
Linearithmic	$O(n \log n)$	Merge sort
Quadratic	$O(n^2)$	Selection sort
Cubic	$O(n^3)$	Shortest path for graphs
Exponential	$O(2^n)$	n-bit cryptographic key
Factorial	$O(n!)$	Travelling Salesperson

6.2 Limits for Complexity

We can use an alternative proof based on the definitions,

$$\lim_{n\to\infty} \frac{T(n)}{f(n)} \geq 0 \text{ and} \neq \infty \qquad \iff T(n) \in O(f(n))$$

$$\lim_{n\to\infty} \frac{T(n)}{f(n)} > 0 \qquad \iff T(n) \in \Omega(f(n))$$

$$\lim_{n\to\infty} \frac{T(n)}{f(n)} > 0 \text{ and} \neq \infty \qquad \iff T(n) \in \Theta(f(n))$$

7 Selection Sort

Selection sort **selects** the smallest element in the array, and swaps it into the front ‘sorted’ portion.

```
def selection_sort(arr, n):
  for pos = 0 to n - 1:
    best = pos
    for i = pos + 1 to n - 1:
      if arr[i] < arr[best]:
        best = i

    swap arr[best], arr[pos]
```

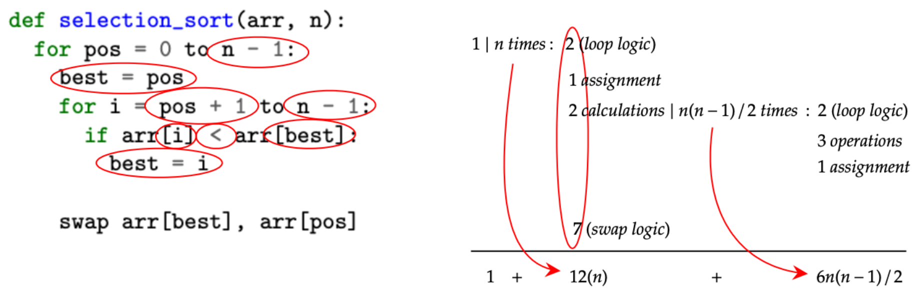
This effectively finds the index of the smallest element (best) and swaps it to the front (pos).

7.1 Linked List Implementation

We can easily translate this code into a linked list implementation.

```
def selection_sort(list):

  if list_empty(list): return

  pos = list.head
  while pos.next != null:
    best = pos
    curr = pos.next
    while curr != null:
      if curr.item < best.item:
        best = curr
      curr = curr.next

    swap best.item, curr.item
    pos = pos.next
```

7.2 Time Complexity



```
def selection_sort(arr, n):
  for pos = 0 to n - 1:
    best = pos
    for i = pos + 1 to n - 1:
      if arr[i] < arr[best]:
        best = i

    swap arr[best], arr[pos]
```

1 | n times :
2 (loop logic)
1 assignment
2 calculations | n(n − 1) / 2 times : 2 (loop logic)
3 operations
1 assignment
7 (swap logic)
1 + 12(n) + 6n(n − 1) / 2

$$\therefore T(n) = 3n^2 + 9n + 1$$
$$T(n) \in O(n^2)$$

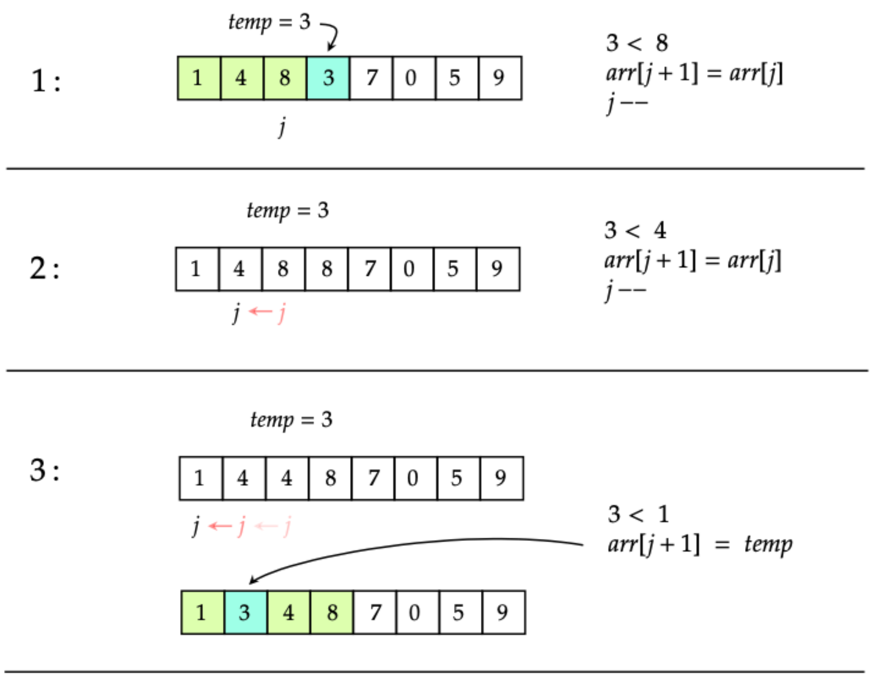
8 Insertion Sort

Insertion sort inserts the next element in the correct spot within the already sorted portion of the array.

```
def insertion_sort(arr, n):
  for next = 1 to n - 1:
    temp = arr[next]
    j = next - 1
```

```
        while j >= 0 and temp < arr[j]:
          arr[j + 1] = arr[j]  # move element one to the right
           ↪  to make space
          j = j - 1
        arr[j + 1] = temp
```

8.1 Example Illustration



8.2 Time Complexity

Worst Case: $T(n) \in O(n^2)$

Best Case: $T(n) \in O(n)$

- this occurs when the array is already sorted, so we don't ever enter the while loop and check the sorted portion of the array.

9 Merge Sort

Merge sort is a divide and conquer recursive algorithm that

- divides the array in two halves

- recursively sorts the two halves
- merge the two halves

9.1 General Algorithm

```
def msort(arr, lo, hi):
  if hi <= lo:
    return

  mid = lo + (hi-lo) // 2
  msort(arr, lo, mid)
  msort(arr, mid + 1, hi)
  merge(a, lo, mid, hi)
```

9.2 Merge Algorithm

The general description of this algorithm is you move left to right through both halves, comparing each element from the halves.

We have two pointers i, j. They start as the first element of each half.

```
i = lo
j = mid + 1
```

We then copy everything to an auxiliary array. This aux array doesn't get sorted, it's just so we have access to the elements after we merge.

```
for k <- lo to hi:
  aux[k] = arr[k]
```

Now we can start the main algorithm. We start with checking if we reach the end of either subarray. With this case, we can copy all the elements of the other half as it will already be sorted!

Otherwise, we simply compare the elements at index i, j and merge!

```
for k <- lo to hi:
  if i > mid:         # end of left half
    arr[k] = aux[j]
    j = j + 1

  else if j > hi:     # end of right half
    arr[k] = aux[i]
    i = i + 1
  else if aux[i] < aux[j]:  # element i is smaller
    arr[k] = aux[i]
    i = i + 1

  else:                 # element j is smaller
    arr[k] = aux[j]
    j = j + 1
```

9.3 Time Complexity

We can use a recursion tree to analyze the time complexity of recursive algorithms.

- nodes are the size of the sub-problem
- sum the amount of work at each problem

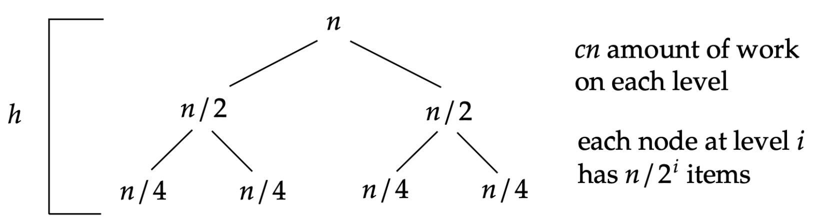
9.3.1 Case 1: n is a power of 2

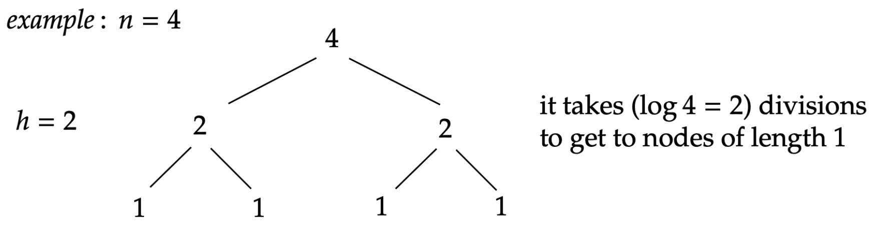
There are two things to account for the time complexity. One is the work done by the merge algorithm, and the work done by the dividing the array in 2.

- each level of the tree has cn operations for the merge algorithm.
- each node at level i, contains $\frac{n}{2^i}$ items in the array, assuming i starts at 0.
- $\therefore$ for the height of the tree, the number of divisions to get the items at level h to equal 1 is represented by

$$\frac{n}{2^h} = 1$$
$$\log(n) = h$$

  This is simply because we keep dividing the array by 2 until there is 1 array item in each node, then we start our merging process. So we assume h is the height where nodes are of size 1, which would take $\log n$ amount of work to get to.


h
n
$n/2$
$n/2$
$n/4$
$n/4$
$n/4$
$n/4$
cn amount of work on each level
each node at level i has $n/2^i$ items

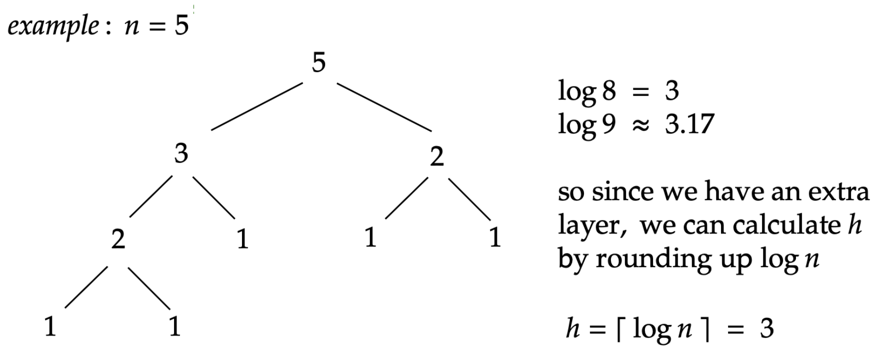

example: $n = 4$
$h = 2$
4
2
2
1
1
1
1
it takes ($\log 4 = 2$) divisions to get to nodes of length 1

So for the best case where n is a power of 2, $T(n) = cn\log(n) \in O(n \log n)$

9.3.2 Case 2: n is not a power of 2

This occurs when the left split is bigger than the right. This adds only 1 layer at most. If n is not a power of 2, when we do $\log n = h$ we get a decimal. For example, $\log 8 = 3$ but $\log 9 = 3.17$. Since this will add one more layer to our tree, we can use the ceiling function!

$$T(n) \leq cn\lceil \log n \rceil \leq c(n \log n + 1) \in O(n \log n)$$


example: $n = 5$
5
3
2
2
1
1
1
1
1
$\log 8 = 3$
$\log 9 \approx 3.17$
so since we have an extra layer, we can calculate h by rounding up $\log n$
$h = \lceil \log n \rceil = 3$

9.3.3 Recurrence Relations

We can also model $T(n)$ as a recurrence relation. We start with the base cases:

$$T(0) = T(1) \approxeq c$$

where the constant c is the **constant** amount of work if the array was empty or of size 1.

For the recursive cases, since we are recursively calling **msort()** twice where we divide the array length by 2 ($n/2$), we can represent $T(n)$ as

$$T(n) = T\left(\lceil \frac{n}{2} \rceil\right) + T\left(\lfloor \frac{n}{2} \rfloor\right) + cn$$

where cn is the work done by the merge algorithm! Note we use the ceiling and floor functions here in case n is not even.

9.4 Space Complexity

- cn for original array
- cn for auxiliary array
- $d \log n$ for call stack

Note, the d represents constant factors like space taken by variables, etc. Therefore, $S(n) \approxeq 2cn + d \log n \in O(n)$

10 Quicksort

Quicksort is a divide and conquer recursive algorithm like mergesort. It sorts in place, so there is no need for an auxiliary array.

- partition the array around a **pivot** element
- left side of the pivot are less than the pivot
- right side is larger than the pivot
- this ensures the pivot is in the correct place within the array
- recursively sort the left and right sides

Note: we don't introduce a way to choose the pivot, but there are ways to choose the best pivot (not talked ab in this class).

10.1 General Structure

```python
def qsort(arr, lo, hi):
  if hi <= lo:
    return
  pivot_index = partition(arr, lo, hi)
  qsort(arr, lo, pivot_index - 1) # left side
  qsort(arr, pivot_index + 1, hi) # right side
```

10.2 Partition Algorithm

This algorithm takes the pivot and 'puts' it in the right place in the array, ensuring the left side is smaller than the pivot, and the right side is larger.

We start with pointers i, j. This assumes our pivot is the first element.

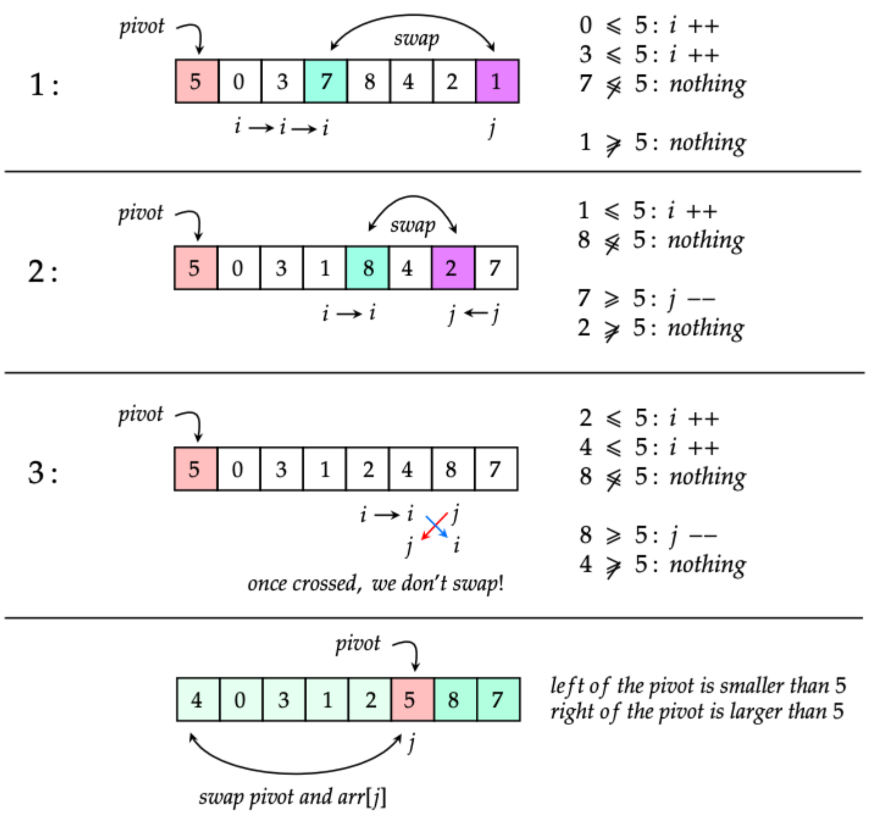
```
i = lo + 1
j = hi
pivot = arr[lo]
```

In our loop, we increment i whenever $arr[i]$ is **SMALLER** than the pivot as we want to keep these items on the left side. We stop if its larger. Similarly, we decrement j whenever $arr[j]$ is **LARGER** than the pivot, and stop when its smaller. Once we've stopped both inner loops, we can swap $arr[i]$ and $arr[j]$. This essentially brings items larger to the right side, and smaller to the left side.

```
loop:
  while arr[i] <= pivot:
    i = i + 1
    if i == hi:
      break

  while arr[j] >= pivot:
    j = j - 1
    if j == lo:
      break

  if i >= j:
    break (infinite loop)

  swap(arr[i], arr[j])

swap arr[lo], arr[j]
return j
```

Once the pointers i, j cross, we no longer swap any items. We can swap the pivot into the index of j and return.

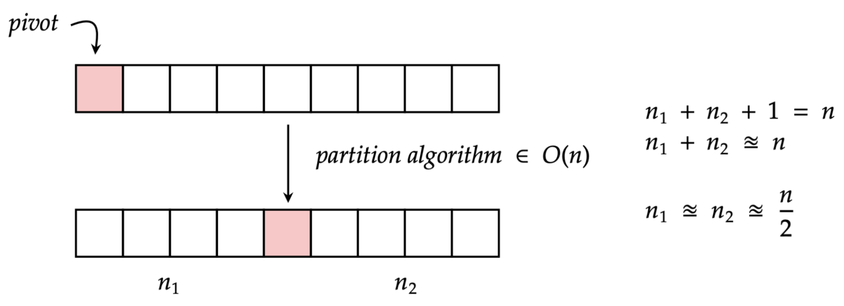
10.3 Example Drawn Out



10.4 Time Complexity

10.4.1 Best Case

The best case is when every partition is even so the 2 subarrays are of same length.


pivot
partition algorithm $\in O(n)$
n_1
n_2
$n_1 + n_2 + 1 = n$
$n_1 + n_2 \approxeq n$
$n_1 \approxeq n_2 \approxeq \frac{n}{2}$

If $n_1 \approxeq n_2 \approxeq n/2$,

$$\begin{aligned}
T(n) &= T(n_1) + T(n_2) + cn \\
T(n) &= T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + cn \\
T(n) &= 2 \cdot T\left(\frac{n}{2}\right) + cn
\end{aligned}$$

Notice how this is the same as merge sort! So we can say $T(n) = 2 \cdot T(\frac{n}{2}) + cn \in O(n \log n)$.

10.4.2 Worst Case

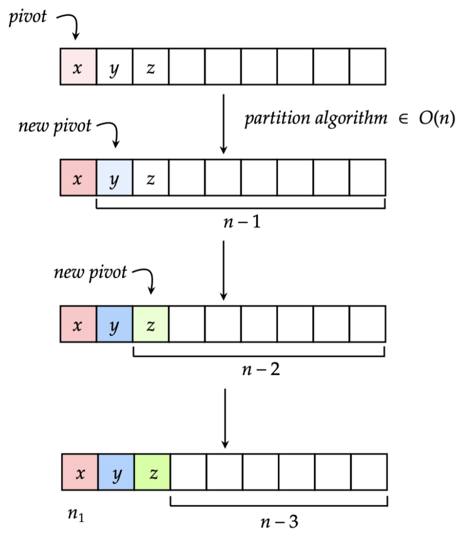
The worst case happens when the array is already sorted or when our pivot is consistently chosen as the smallest element. After every partition, the pivot stays where it is since it is the smallest element of the subarray. This makes our recurrence relation the following,

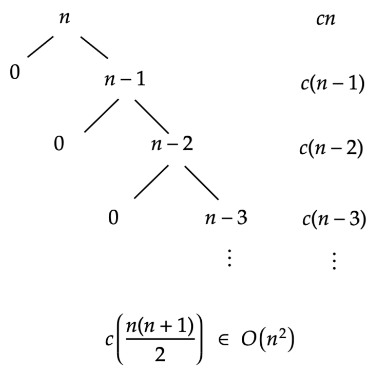
$$\begin{aligned}
T(n) &= T(0) + T(n-1) + cn \\
T(n) &= T(n-1) + cn
\end{aligned}$$

Going through the entire tree we can see that,

$$T(n) = c\left(\frac{n(n+1)}{2}\right)$$

which means $T(n) = c\left(\frac{n(n+1)}{2}\right) \in O(n^2)$.





10.4.3 Almost Worst Case

Say the partition always splits the array 90% on one side, 10% on the other. Here $n_1 = \frac{n}{10}$ and $n_2 = \frac{9n}{10}$. We can model the recurrence relation as

$$T(n) = T\left(\frac{n}{10}\right) + T\left(\frac{9n}{10}\right) + cn$$

Modeling this as a tree we can see,

On the left side (10%), we can find h with the following,

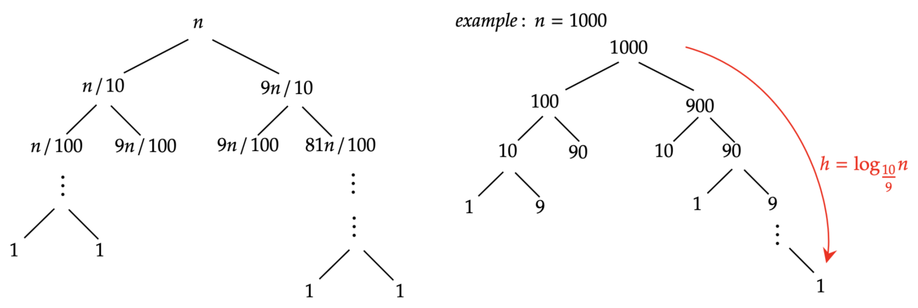
$$\begin{gathered}\frac{n}{10^h} = 1\\ n = 10^h\\ \log n = h \log 10\\ \frac{\log_2 n}{\log_2 10} = h\\ \log_{10} n = h\end{gathered}$$

However, the right side (90%) is always deeper, so we calculate that with

$$\begin{gathered}\frac{9n}{10^h} = 1\\ \log 9n = h \log 10\\ \frac{\log_2 9n}{\log_2 10} = h\\ \log_{\frac{10}{9}} n = h\end{gathered}$$

Therefore,

$$T(n) \leq cn \log_{\frac{10}{9}} n \in O(n \log n)$$



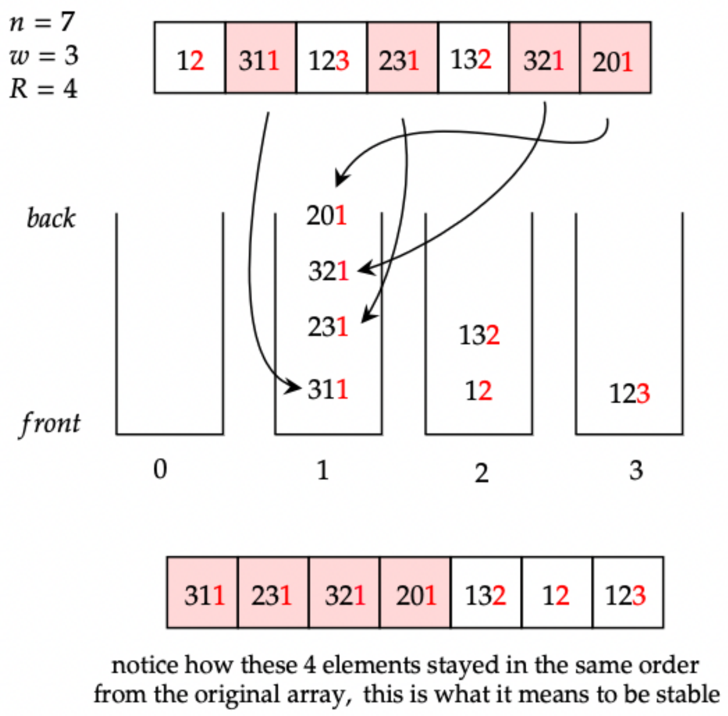
11 Radix Sort

Radix sort compares the least significant digits (LSD) of the numbers, and sorts based on that. For example, [19, 20, 5] would sort 9, 0, 5 on the first pass.

- n integers (size n)
- w digits in the integer
- R – radix, more commonly known as the base of the digit (e.g. decimal is radix 10, binary is radix 2)

11.1 Algorithm Structure

We start by creating an array of R ‘buckets’ which are queues. We use queues to maintain stability. For each item in the array, we look at its LSD and put it in its respective bucket. Once all the numbers are in their buckets, we dequeue them in order and repeat.



$n = 7$
$w = 3$
$R = 4$
12
311
123
231
132
321
201
back
front
201
321
231
311
132
12
123
0
1
2
3
311
231
321
201
132
12
123
notice how these 4 elements stayed in the same order from the original array, this is what it means to be stable

This is what the algorithm looks like in code:

```
def radix_sort(arr, R, w):
  buckets = array(R) of Queues # array of R number of queues

  for p = 0 to w - 1:  # for each sig dig

    # putting element into corresponding bucket
    for item in arr:
      d = item % R^(p+1) // R^p
      enqueue(buckets[d], item)

    # taking element out and sorting
    i = 0
    for bucket in buckets:
      while not is_empty(bucket):
        a[i] = dequeue(bucket)
        i = i + 1
```

The logic behind $d = item \ \% \ R^{p+1} \ // \ R^p$ is shown with an example:

$$\begin{aligned}
R = 10, w = 4, p = 1 \\
item = 3758 \\
d = item \ \% \ R^{p+1} \ // \ R^p \\
d = 3758 \ \% \ 10^2 \ // \ 10^1 \\
d = 3758 \ \% \ 100 \ // \ 10 \\
d = 58 \ // \ 10 \\
d = 5
\end{aligned}$$

11.2 Changing Radix Base R

To use a different number of buckets, we can use the following:

- max number in radix $R = R_1^{w_1} - 1$
- $w_2 = \lceil \log_{R_2}(\max) \rceil = \lceil \frac{\log \max}{\log R_2} \rceil$

Here is how we derived w_2

$$\begin{aligned}
R_2^{w_2} &= 9999 \\
w_2 \log R_2 &= \log 9999 \\
w_2 &= \log 9999 / \log R_2 + 1
\end{aligned}$$

11.2.1 Example

Change $R_1 = 10, w_1 = 4$ to $R_2 = 8$,

$$\begin{aligned}
\max &= R_1^{w_1} - 1 \\
\max &= 10^4 - 1 = 9999 \\
w_2 &= \lceil \log_8(9999) \rceil = 5
\end{aligned}$$

11.3 Time Complexity

For radix sort $T_{best}(n) = T_{avg}(n) = T(n)$. It is based on 4 things:

- creating R buckets ($c_2 R$)
- w outer loops for each digit
- n items going into the buckets ($c_1 n$)
- n items removed from the buckets ($c_1 n$)

Therefore our time complexity looks like the following function:

$$\begin{aligned}
T(n) &\approxeq (c_1 n + c_1 n) \cdot w + c_2 \cdot R \\
T(n) &\approxeq 2c_1 nw + c_2 R
\end{aligned}$$

If R is held constant, then $T(n) \in O(n)$. However, note that as $w \to n$, $T(n) \to O(n^2)$.

12 Questions

Here are the most common types of questions and the method to solve them!

12.1 Algorithm Analysis

If you have prove $T(n) \in O(f(n))$, you can use the definition, bump terms up, and prove. For **proving by definition**, you need a pre-proof analysis and a formal proof:

- **pre-proof analysis**: finding values for c, n_0
- **formal proof**: with values for c, n_0, work backwards to show $T(n) \leq c \cdot f(n)$

12.1.1 Example:

Prove $35n^3 - 150n + 1 \in O(n^3)$ Start with the pre-proof analysis. By definition,

$$35n^3 - 150n + 1 \in O(n^3)$$
$$35n^3 - 150n + 1 \leq 36n^3 \text{ for all } n \geq 2$$
$$c = 36, n_0 = 2$$

Now for the formal proof. Choose $c = 36, n_0 = 2$

$$0 \leq 35n^3 - 150n + 1 \text{ for all } n \geq 2$$
$$-150n + 1 \leq n$$
$$n \leq n^3$$
$$35n^3 - 150n + 1 \leq 35n^3 + (n^3)$$
$$35n^3 - 150n + 1 \leq 36n^3 = cn^3$$
$$\therefore 35n^3 - 150n + 1 \in O(n^3)$$

If you have a question asking to prove $T(n) \in \Omega(f(n))$ or $T(n) \notin O(f(n))$, its best to use limits. For $T(n) \notin O(f(n))$, you want to use proof by contradiction and show that $\infty > c$, otherwise for $T(n) \notin \Omega(f(n))$, you show the limit is > 0.

12.1.2 Example:

Prove $20n^2 - 33n - 22 \in \Omega(n^2)$.

$$\lim_{n\to\infty} \frac{T(n)}{f(n)} > 0$$
$$\lim_{n\to\infty} \left(\frac{20n^2 - 33n - 22}{n^2}\right) > 0$$
$$\lim_{n\to\infty} \left(\frac{n^2(20 - \frac{33}{n} - \frac{22}{n^2})}{n^2}\right) > 0$$
$$\lim_{n\to\infty} 20 > 0$$

Note if this was $T(n) \notin O(f(n))$, we would use the formal definition + limits:

$$20n^2 - 33n - 22 \leq cn^2$$
$$\lim_{n\to\infty} \left(\frac{20n^2 - 33n - 22}{n^2}\right) > c$$
$$20 > c$$

Therefore by contradiction, we can prove n^3 is not bounded above $T(n)$ for $c < 20$.

What if the question asks to prove $T(n) \in \Omega(f(n))$ by definition? There are two cases, one where the degree of $T(n)$ is the same as $f(n)$ like in our example above. Here is the formal proof for that. We can assume $c = 19$ (sort of cheating since we know $c < 20$ for it to be bounded below).

$$-33n - 22 \geq -(n^2) \text{ for all } n \geq 34$$
$$20n^2 - 33n - 22 \geq 20n^2 - n^2$$
$$20n^2 - 33n - 22 \geq 19n^2 = cn^2$$
$$20n^2 - 33n - 22 \in \Omega(n^2)$$

Methods for proving $T(n) \notin \Omega(f(n))$ are similar to $T(n) \notin O(f(n))$, however, with different powers we use the same method as before.

12.1.3 Example:

Prove $11n^2 - 43 \notin \Omega(n^3)$. Here we are saying n^3 is not bounded below $11n^2 - 43$. We can do 2 methods:

$$\begin{gathered} T(n) < cn^3 \\ 11n^2 - 43 < 11n^2 \\ 11n^2 < 11n^3 \text{ for all} n \geq 2 \\ 11n^2 - 43 < 11n^3 \\ \therefore c = 11, n_0 = 2 \end{gathered}$$

and continue with the formal proof. Here is how the professor does it,

$$\begin{gathered} 11n^2 - 43 < 11n^2 \\ 11n^2 < cn^3 \\ \frac{11}{c} < n \end{gathered}$$

Formal Proof: For every $c > 0, n_0 \geq 0$, if we choose an n such that $n \geq n_0, n \geq 1, n \geq \frac{11}{c}$

$$\begin{gathered} n > \frac{11}{c} \\ cn^3 > 11n^2 \\ 11n^2 - 43 < 11n^2 \\ (11n^2 - 43 < 11n^2) \wedge (11n^2 < cn^3) \\ 11n^2 - 43 < cn^3 \\ \text{Theorem: Transitivity of } < \end{gathered}$$

Since c and n_0 are arbitrarily chosen, it is proven.

12.1.4 Example:

Prove $n^2 + n - 555 \notin O(n)$. Here we are saying n is not bounded above $n^2 + n - 555$. To prove this via definition, we use the professors method:

Pre-proof analysis: We need to find an n to show $n^2 + n - 555 > cn$.

$$n^2 + n - 555 > n^2 \text{ for any } n \geq 555$$

because its easy to show that

$$n^2 > cn$$

when $n > c$.

Formal Proof: Using the definition of $T(n) \notin O(n)$, we say for every $c > 0, n_0 \geq 0$, we choose an n such that $n \geq n_0, n \geq 555, n > c$.

$$n > c$$
$$n^2 > cn$$
$$n^2 + n - 555 > n^2$$

Since $n^2 + n - 555 > n^2$ and $n^2 > cn$, we can say

$$(n^2 + n - 555 > n^2) \wedge (n^2 > cn)$$
$$n^2 + n - 555 > cn$$
$$n^2 + n - 555 \notin O(n)$$

which is the theorem of transitivity! Since c, n_0 are arbitrarily chosen, it is proven.

12.1.5 Summary

- if same degree: $\in O(f(n))$ and $\in \Omega(f(n))$ bump up/down terms
- if not: $\notin O(f(n))$ and $\notin \Omega(f(n))$ use transitivity.