COSC160: Data Structures Linked Lists

Jeremy Bolton, PhD
Assistant Teaching Professor

GEORGETOWN UNIVERSITY

Outline

I. Retrieval or Search in List Structures.

II. Imposing Order
    I. Order by frequency of access
        I. Using Expected Values for Average Case Analysis
        II. Order Heuristic: Move-To-Front
    II. Order by Value
        I. Another sorting example

GEORGETOWN UNIVERSITY

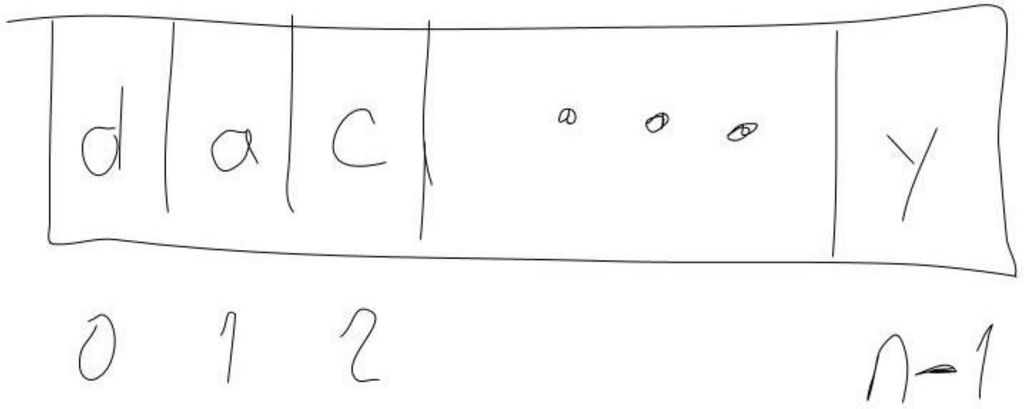
Searching

- When searching a list for an item, it may be necessary to traverse the entire list. We will investigate two simple speed-up schemes:
  - We can improve search time with improved <u>organization</u> of the data in our structure
    1. Order based on data value
    2. Order based on frequency of access

list.search(item)

Search List: Worst and Average Case

- In general, sequential search, both Worst and Average Cases are $\Theta(n)$
- Can we improve upon this? Yes – with some assumptions / constraints.
  - Keeping our data ordered (organized) can lead to speed-ups.



Frequency Based Analysis

- Order based on frequency of access.

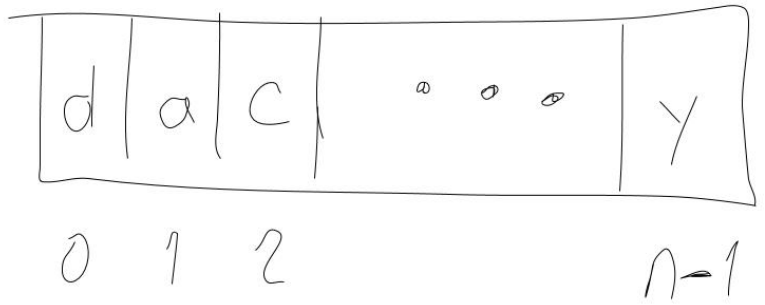
- Scenario: Assume you are designing a data structure to store records for an institution. Records are retrieved from time to time. Some records may be retrieved more often than others. Some records may be retrieved almost never.

- Lets re-inspect the average case…

Average Case as Frequency Based Analysis

- Average Case: Add up steps for all cases and divide by number of a cases.

- Average number of comparisons $\frac{1+2+3+\cdots+n}{n}$ is $\Theta(n)$



Using Expected Values for Average Case

- Note an average is similar to taking the *expected value* (with the probability of each case being equally likely: $\frac{1}{n}$ for the n cases here). That is assume the step count is a random variable of each case and has some probability of being observed.

- $E[count] = \Sigma_{i=1}^{n} count_{case_i} * p_{uniform}(case_i) =$

$$(1)\frac{1}{n} + (2)\frac{1}{n} + (3)\frac{1}{n} + \ldots + (\mathrm{n})\frac{1}{n} = \frac{1+2+3+\cdots+n}{n}$$

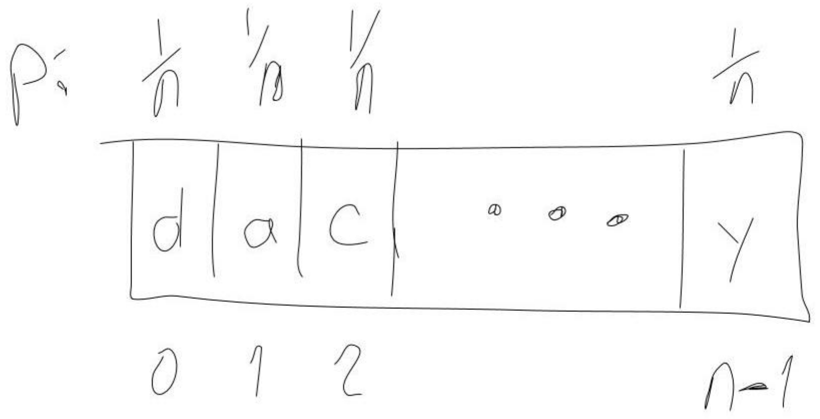
where $p_{uniform}(case_i) = 1/n$ for all n cases is the probability of $case_i$. (A uniform distribution each case is equally likely)

If each case is equally likely then this will give us a good estimate of the expected (average) comparison count. However, if each case is not equally likely, we can better estimate the expected count using the true probability of occurrence of each case.

Expected (Average Case) Comparison Count

- Standard Average (Uniform) Example:

$$\Sigma_{i=1}^{n} count_{case_i} * p(case_i) = \Sigma_{i=1}^{n} i * \frac{1}{n} = \frac{1}{n}\Sigma_{i=1}^{n} i = \boxed{\frac{n+1}{2}} = \Theta(n)$$

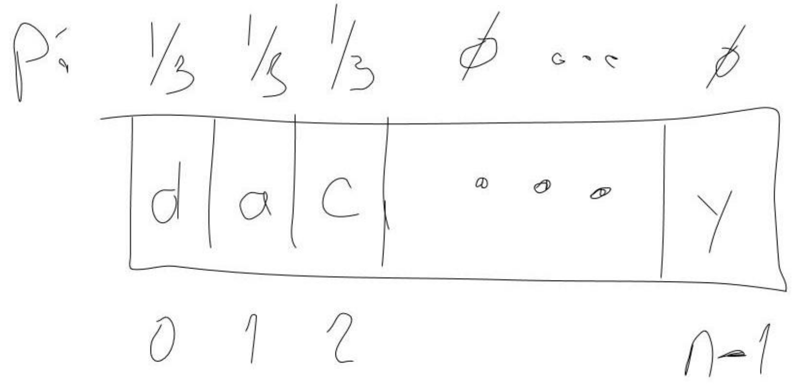


Using Expected Value to Compute Average Case

- Extreme Example: Lets assume case 1, 2, and 3 are likely to be searched with probabilities of 1/3 each. And all other records have a probability of zero of being searched.

- What would happen if we placed items with higher probability of search near the front of the list?

- $\Sigma_{i=1}^{n} count_{case_i} * p(case_i) = (1)\frac{1}{3} + (2)\frac{1}{3} + (3)\frac{1}{3} + (4)0 + \cdots + (n)0 = \boxed{2} = \Theta(1)$



Ordering a list based on frequency of access

- Intuitively, it would then be best to order the items based on their probability of search.

- Good in theory! However, one generally does not know this probability distribution for items searched.

- If we knew the probability of access of each item, how would we order the items?
  - Increasing order of probability

Order based on frequency of access

- Heuristic approach: every time a record is accessed (searched and found), place that item at the beginning of the array. The items that are searched most often, should have a high probability of being near the beginning of this list (thus reducing search time)
  - AKA: MOVE TO FRONT

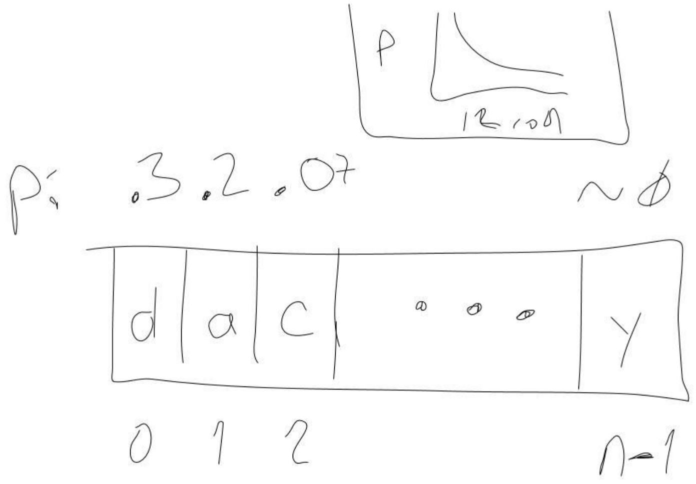
- Implementation Details:
  - Would you implement this structure as a linked list or an array? Why?

Using Expected Value to Compute Average Case

- Another Example: Lets assume the different cases have a probability of being searched that resembles a Poisson distribution. And lets assume that the items just happen to be in decreasing order of probability of being searched.

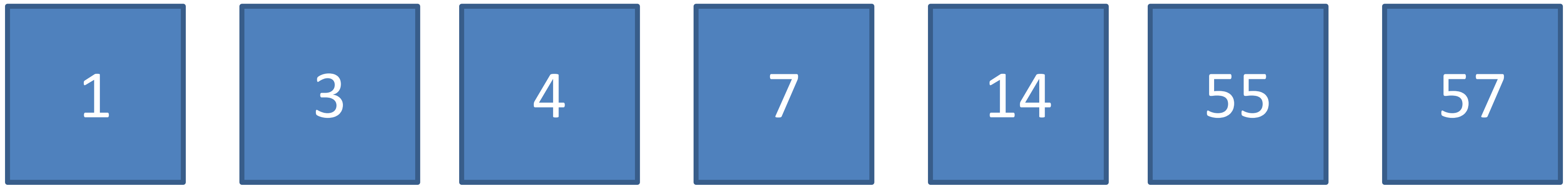
- $\Sigma_{i=1}^{n} count_{case_i} * p_{Poisson}(case_i) \cong \boxed{2} = \Theta(1)$

- Note this is a very "steep" distribution and the effectiveness of this ordering method will drastically depend upon the true search distribution of the data



Order based on data value

- If the data are ordered, we can reduce search time
  - Numeric order, alphabetical order, binary order, …
  - E.G. files in a filing cabinet
- Binary Search
  - Divide and conquer scheme
  - Example: Search for 55



Pseudocode for Binary Search (iterative)

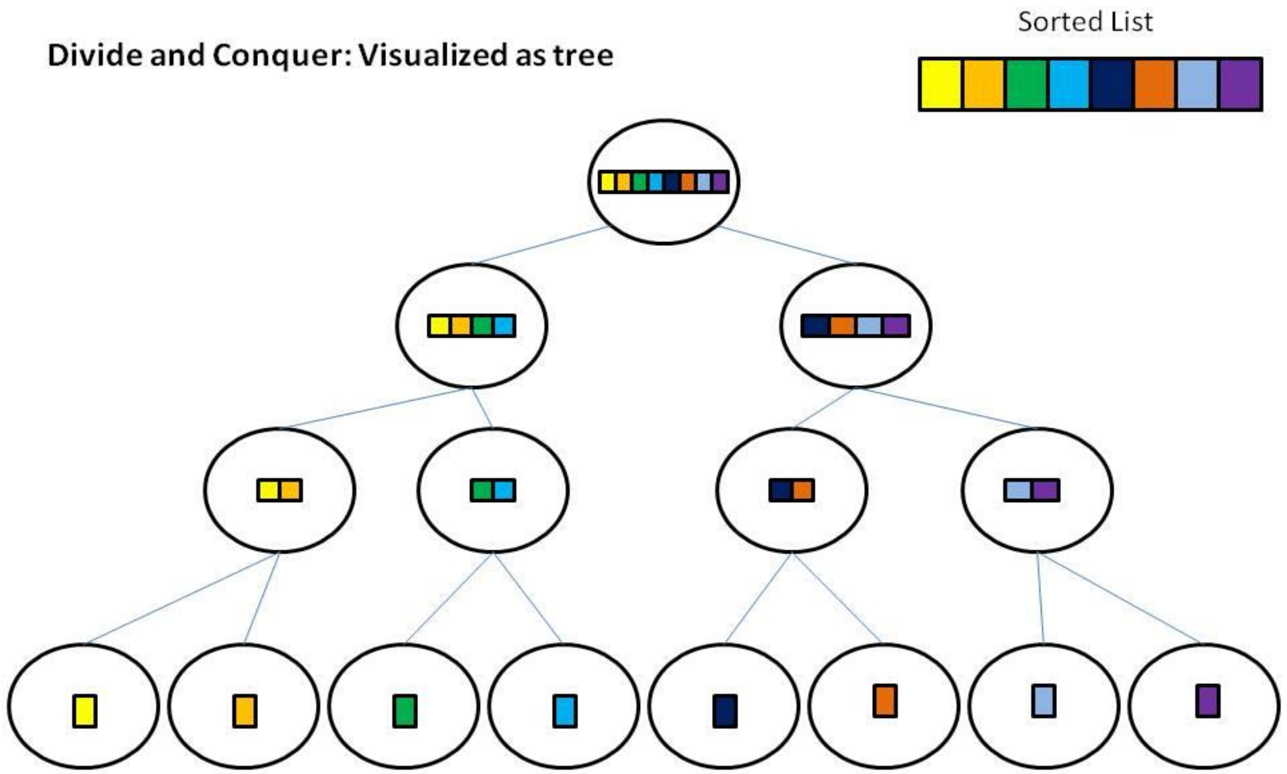
- Worst Case Time Complexity?

```
// Searches using DaC for x in a where a is sorted.                 Step Count
// Returns index of x, else -1
Algorithm BinarySearch(a, n,x){
  index = -1;                                                        1
  searchBegin = 1;                                                   1
  searchEnd = n;                                                     1

  while searchBegin <=  searchEnd do{
          searchMid = floor((searchEnd + searchBegin)/2);            ?
          if x > a[searchMid]{                                       ?
                 searchBegin = searchMid+1;                          ?
          } else if x < a[searchMid]{                                ?
                 searchEnd = searchMid-1;                            ?
          } else if x == a[searchMid]{                               ?
                 index = searchMid;                                  ?
          } end if
  }end while
return index;                                                        1
}end
```

Binary Search: decision tree diagram

- The flow of execution is represented as a single path from root to leaf, where each step from parent to child represents 1 iteration in the binary search method.
- Thus the number of iterations can be bounded by the height of this tree.
  - Note that the number of nodes at each level of our tree doubles at each depth of the tree.
  - Nodes at depth d = ?
  - How many nodes are at the lowest level of this tree?



Using Recurrences: Binary Search Time Complexity

- Note:
  - The size of the list to be searched is halved during each recursive call.
  - During each call we perform about 3 comparison checks. We will simply use c to indicate some constant
- Define recurrence T(n): Comparison count of binary search for an ordered list of size n

T(n) = 3 + T(⌊n/2⌋) , where T(1) = 1

Using backward iteration, we will show T(n) is logarithmic for $n = 2^j$.

T(n) = 3 + T(⌊n/2⌋)
= 3 + 3+ T(⌊n/4⌋)
= 3 + 3+ T(⌊n/8⌋)
…
= 3 + 3+ …. + T(1)

$$= 1 + \sum_{i=1}^{\log_2 n} 3 = 1 + 3\log_2 \text{n}$$

Observe Pattern / Series:
How many times can we divide n by 2 before reaching our boundary case?

$j = \log_2(n)$, or similarly $2^j = n$

Try at home: Use substitution method, (induction) prove that T(n) is O(log n)

GEORGETOWN UNIVERSITY

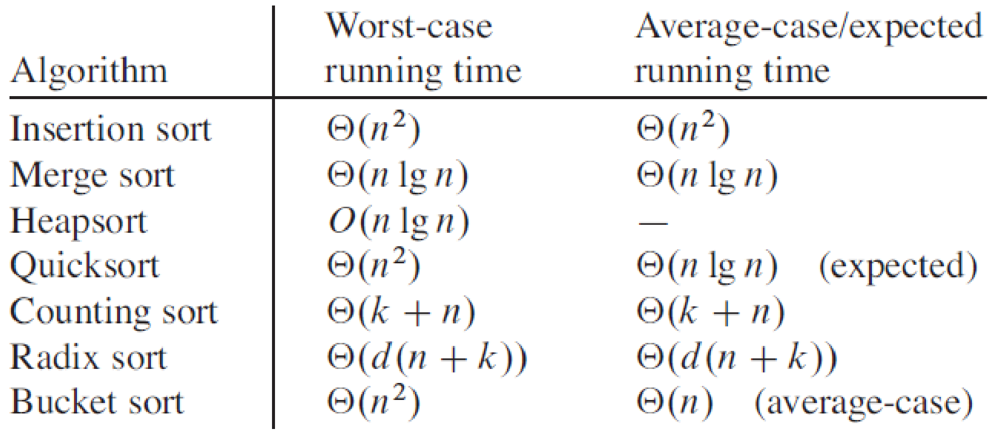
Binary Search vs Sequential Search

- With the added constraint of order we can reduce search time from $\Theta(n)$ to $\Theta(\log_2 n)$

- Implementation Details:
  - Would you use a binary search scheme for an array? For a linked list? Why or why not?

Sorting: Imposing Order

- Insertion Sort
  - Intuitive and in-place
  - But quadratic time complexity in the worst case
- Merge Sort
  - Uses a “Divide and Conquer” Scheme (similar to binary search) to improve efficiency
- We will investigate a few sorting algorithms throughout the course.

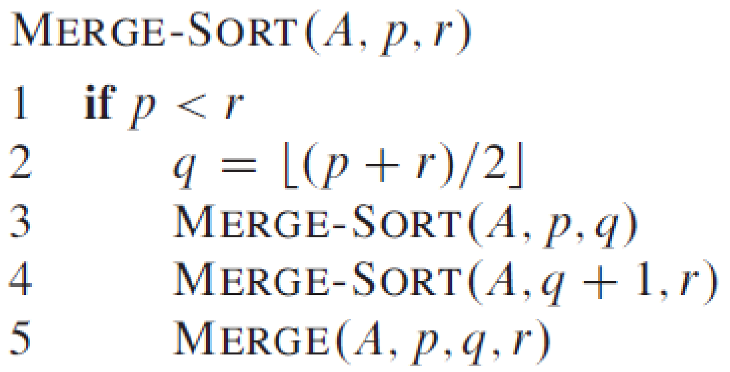


Algorithm	Worst-case running time	Average-case/expected running time
Insertion sort	$\Theta(n^2)$	$\Theta(n^2)$
Merge sort	$\Theta(n \lg n)$	$\Theta(n \lg n)$
Heapsort	$O(n \lg n)$	—
Quicksort	$\Theta(n^2)$	$\Theta(n \lg n)$ (expected)
Counting sort	$\Theta(k + n)$	$\Theta(k + n)$
Radix sort	$\Theta(d(n + k))$	$\Theta(d(n + k))$
Bucket sort	$\Theta(n^2)$	$\Theta(n)$ (average-case)

GEORGETOWN UNIVERSITY

Merge Sort

- Inputs (practical characterization of domains):
  - A: an array of ints of length n
  - p: index (begin)
  - q: index (end)
- Assumptions:
  - Precondition: p = 1 AND q = n
- Correctness
  - Postcondition: elements in A are in increasing order

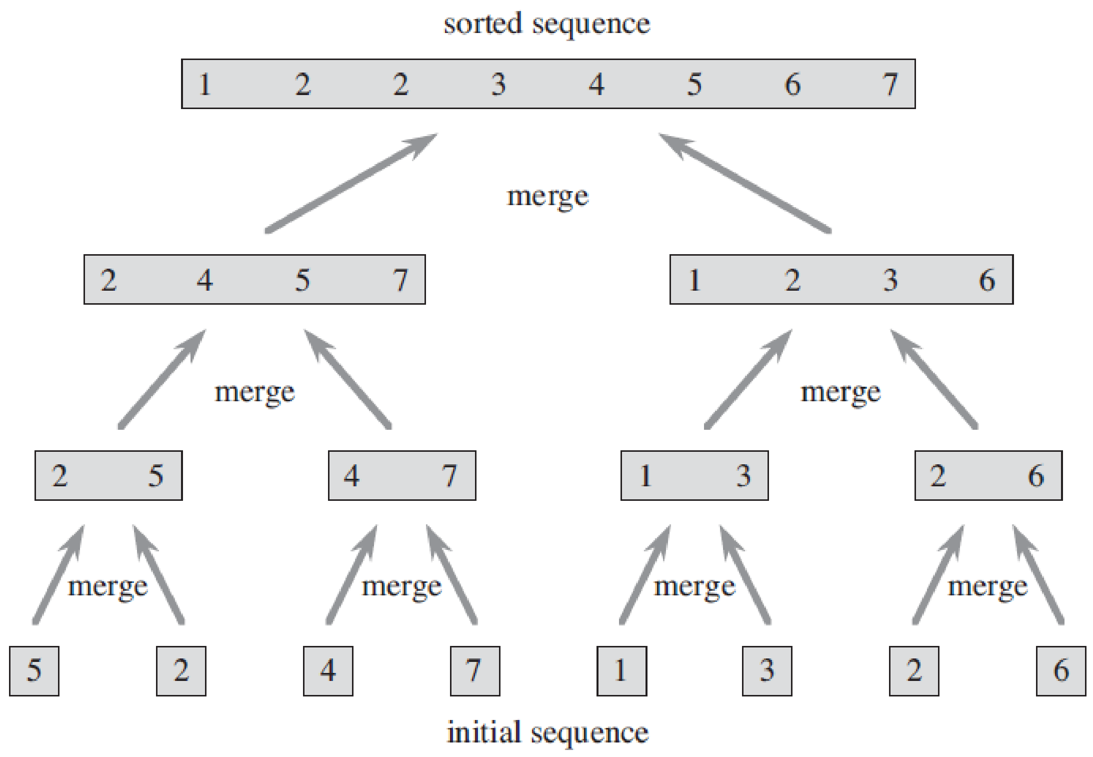


```
MERGE-SORT($A, p, r$)
if $p < r$
    $q = \lfloor (p + r)/2 \rfloor$
    MERGE-SORT($A, p, q$)
    MERGE-SORT($A, q + 1, r$)
    MERGE($A, p, q, r$)
```

Merge Sort

```
MERGE-SORT(A, p, r)
1  if $p < r$
2      $q = \lfloor (p + r)/2 \rfloor$
3      MERGE-SORT(A, p, q)
4      MERGE-SORT(A, q + 1, r)
5      MERGE(A, p, q, r)
```

- Intuition
  - Repeatedly divide array in halves until there are multiple arrays of length 1.
    - this will take log time
  - Repeatedly merge these arrays.
    - The merge requires a sequential scan of each subarray, which will be linear in total



Merge Sort

- The worst case time complexity of Merge Sort is O(n log n)
- Proof:
  - Define step count recurrence
  - T(n) = T(n/2) + T(n/2) + c + M(n/2+n/2)
    - c is the constant number of steps needed to check for the base case, compute q, …
    - M() is the number of steps needed to merge two subarrays each of length n/2, which can be done in linear time using a sequential scan
    - Thus …

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

$$= 2\left[2T\left(\frac{n}{4}\right) + c + n\right] + c + n \ldots = 4\mathrm{T}\left(\frac{\mathrm{n}}{4}\right) + 3\mathrm{c} + 3\mathrm{n}$$

$$= 2[4\mathrm{T}\left(\frac{\mathrm{n}}{8}\right) + c + n] + 3\mathrm{c} + 3\mathrm{n} = 8\mathrm{T}\left(\frac{\mathrm{n}}{8}\right) + 5\mathrm{c} + 5\mathrm{n}$$

…

Observe the pattern. We will reach the boundary condition when i = log n

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

…. Note: $\mathrm{i} = \log_2 \mathrm{n}$ at the boundary condition T(1)

$$= \mathrm{nT}(1) \quad + (2\mathrm{i} - 1)\mathrm{c} + (2\log_2\mathrm{n} - 1)\mathrm{n}$$

Which is O(n log n)

GEORGETOWN UNIVERSITY