CRICOS PROVIDER 00123M

School of Computer Science

COMP SCI 1103/2103 Algorithm Design & Data Structure

Sorting algorithms

adelaide.edu.au

seek LIGHT

Previously on ADDS

- Insertion Sort
  - Complexity
    - Worst and average-case $O(n^2)$
- Selection Sort
  - Complexity
    - Worst and average-case $O(n^2)$
- Bubble Sort
  - Complexity
    - Worst and average-case $O(n^2)$
- Quicksort
  - Complexity
    - Worst $O(n^2)$
    - Average-case $O(n \log n)$

Can you give me a new algorithm?

- Remember T(n)=2T(n/2) +cn?
  - We saw this for best case complexity of quick sort
  - Recursive calls take care of sorting sublists.
    - We assume that they are sorted after the recursive call.
  - This gives us O(n log n)

- How else can we sort with the following steps?
  - Splitting the list to two equal size sublists
  - Assume that they are sorted (we will do it recursively)
  - Combine the results in linear time (cn)

Merge Sort

```
function mergesort(list m){

  if length(m) <= 1                              // for a single element
     return m                                    // regard as sorted

  list left, right, result
  middle = length(m)/2                           // find the middle point

  for each x in m up to middle                   // add to the left list
     add x to left
  for each x in m from middle to the end         // add to the right list
     add x to right

  left = mergesort(left)                         // recursive sort
  right = mergesort(right)

  result = merge(left, right)                    // compare and merge

  return result
}
```

Merge Sort

```
function merge(left, right){
    result = empty list

    while left is not empty and right is not empty do
        if first(left) <= first(right) then
            append first(left) to result and remove
        else
            append first(right) to result and remove

    // Either left or right list may have elements left
    while left is not empty do
        append first(left) to result and remove

    while right is not empty do
        append first(right) to result and remove

    return result

}
```

Merge Sort

- Merge sort uses a divide-and-conquer approach.
- What would happen if we had two (smaller) sorted lists?
- The final sort would require us to merge the two sorted lists
  - Compare the head of the lists
  - Place the smaller element in the resulting list
  - Remove that from the given list that it belongs to
  - Go back to comparing the heads until one list is empty
  - Place all elements of the remaining list in the resulting list

- Base case: lists of size one

Merge Sort –recursive version

- We start with a list of length n and split it into two, which will be merged once they are sorted.
- To sort each sub-list, divide into two that can be merged once they are sorted.
- Continue recursively until we hit the single element base condition.
- Then we unwind and merge everything.

Merge Sort

- Performance
  - Worst case is O(n log n)
  - Average case is O(n log n)
  - Best case is O(n log n)

Quicksort vs Merge Sort

- Compared with quick sort
  - Quicksort has a worse complexity in worst case
    - But O(n^2) happens with very low probability
  - Quick sort does not need to keep an extra memory block
  - Quicksort is not stable!

Sorting Algorithms

- Insertion Sort
  - Complexity
    - Worst and average-case $O(n^2)$
- Selection Sort
  - Complexity
    - Worst and average-case $O(n^2)$
- Bubble Sort
  - Complexity
    - Worst and average-case $O(n^2)$
- Quicksort
  - Complexity
    - Worst $O(n^2)$
    - Average-case $O(n \log n)$
- Merge Sort
  - Complexity
    - Worst and average-case $O(n \log n)$

Comparison Sort

- Most of what we've been looking at have been comparison sorts
- The fundamental engine of the sort is comparing two values.
- Comparison sorts include:
  - Quicksort
  - merge sort
  - bubble sort
  - insertion sort
  - selection sort
  - Heapsort (we will cover it at the end of the semester)
- How good can the comparison sort be? $\Omega(n \log n)$

Typical causes of complexity

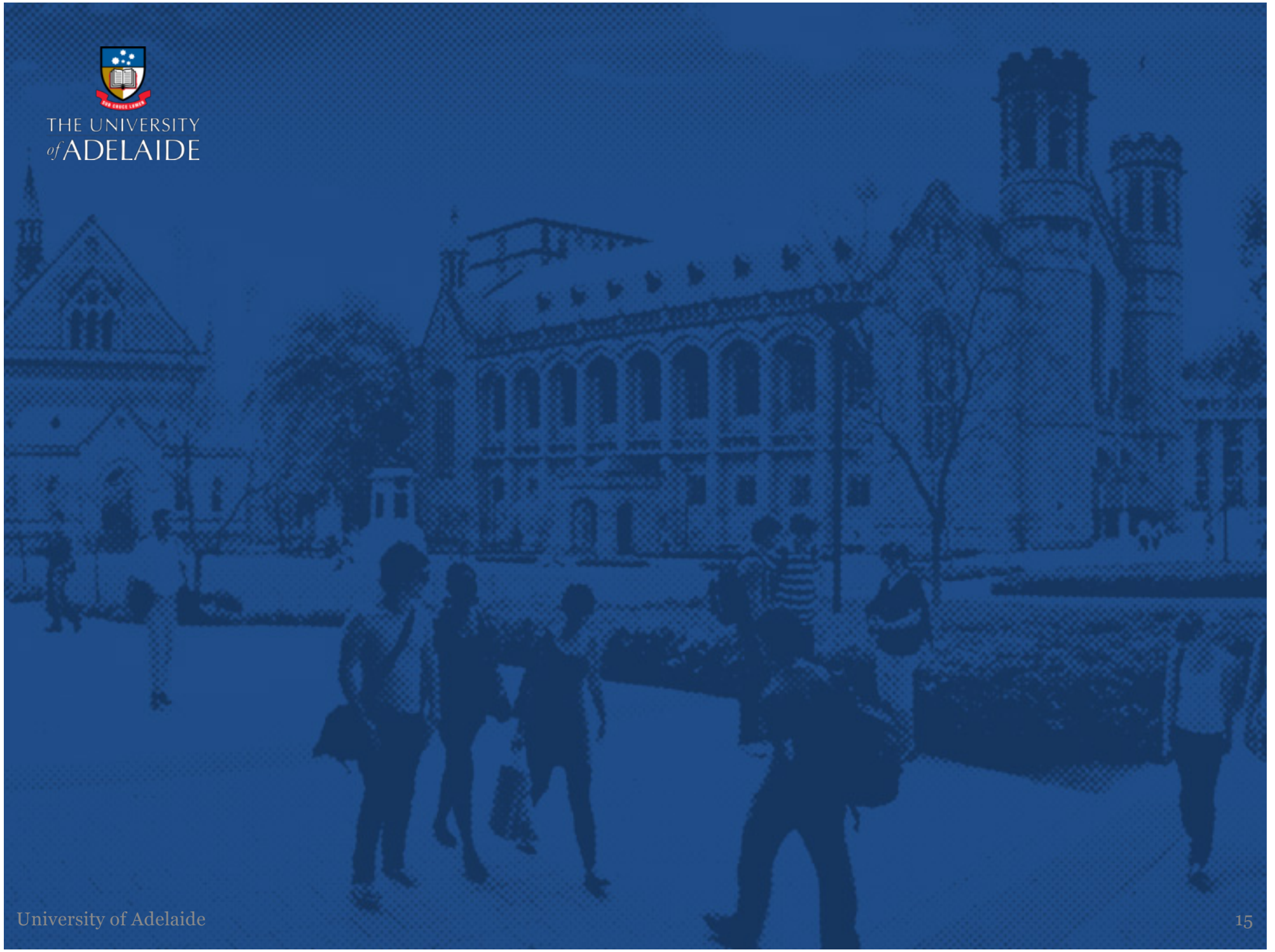
- There are some typical algorithmic structures that lead to common growth rates and data structures/algorithms with well known upper bounds on their growth rates.
- O(1) – a number of statements independent of input size (Hash Tables, with perfect hash function)
- O(n) - loop over size n input (Linear Search)
- $O(n^c)$ - nested loops over size n input (Selection Sort, Insertion Sort)
- O(logn) -repeatedly divide input in half (Binary Search)
- O(nlogn) - “divide and conquer” split input in half and recursively call function until O(1) function can be performed, combine returned values in O(n) (MergeSort)
- $O(c^n)$ - multiple possible combinations of input (or Permutations of a string)

Distribution sort

- We also have another kind of sort- distribution sort!
  - Counting sort
    - O(m+n), m being the number of possibilities for the values
  - Bucket sort
- Basic outline of bucket sort:
  - Set up some empty buckets
  - Go over your original list and scatter the objects into the buckets.
  - Sort each bucket
  - Visit the buckets in turn and gather the results

Bucket sort

- Map the items to buckets (a hash function)
  - The buckets make up a contiguous set - you can think of each bucket as holding a subset of the possible values.
  - Everything that you're sorting will go into one (and only one) of the buckets.
  - Your scatter operation is going to use some sort of fast calculation to work out which bucket things go into.
  - Each item takes O(1), whole scattering O(n)
- The gather operation is a simple one:
  - Walk the buckets, in order
  - Extract the sorted list from each bucket
  - Join them together in sequence


THE UNIVERSITY
of ADELAIDE