School of Computer Science

COMP SCI 1103/2103 Algorithm Design & Data Structure

Searching + more complexity examples

Overview

- Binary search
- Analysis of recursive algorithms: simple methods in this course
- Examples for finding Complexity
  - Fibonacci
  - GCD
  - Binary Search

Complexity Analysis

- How to prove things:
  - If we want to prove Big-oh (and other) notation, the only thing we can rely on is the formal definition.
  - Sometimes, if we want to disprove some statement, at least one counterexample will work.
- We will see examples later.

Complexity Analysis

- How to analyze recursive functions
- It can be quite complicated.
- If the recursion is really just a thinly veiled loop, the analysis is usually trivial

```
int fac(int n){

  if(n <= 1){
    return 1;
  }else{

    return n*fac(n-1);
  }
}
```

T(1)=c'
T(n)=c+T(n-1)
Gives us nc+c'= O(n)

Example

- However, when more than one recursive call is done in the function, it is difficult to convert the recursion into a simple loop structure.
- Recursive Fibonacci has a growth rate of:
- $1/\sqrt{5}\ (\ ((1+\sqrt{5})/2)^n - (1-\sqrt{5}/2)^n)$
- We can show that it is in $\Omega(2^{n/2})$

```
int fib(int n){

  if(n<=1)
    return 1;
  else
    return fib(n-1)+fib(n-2);
}
```

Fibonacci Recursion

Lower Bound:

Assume n is even: $T(0) = T(1) = 1$

$$
\begin{aligned}
T(n) &= T(n-1) + T(n-2) + c \\
&\geq 2 \cdot T(n-2) \\
&= 2(T(n-3) + T(n-4)) \\
&\geq 2 \cdot 2 \cdot T(n-4) \\
&\geq 2^k \cdot T(n-2k) \\
&\geq 2^{(n-2)/2} \cdot T(2) \\
&\geq 2^{(n-2)/2} \cdot 2 \\
&= 2^{n/2} \\
&= \Omega(2^{n/2})
\end{aligned}
$$

Fibonacci Recursion

Upper Bound:

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

$$\begin{aligned}
T(n) &= T(n-1) + T(n-2) + c \\
&\leq 2 \cdot T(n-1) + c \\
&= 2(T(n-2) + T(n-3) + c) + c \\
&\leq 2 \cdot 2 \cdot T(n-2) + 2c + c \\
&\leq 2^k \cdot T(n-k) + c(2^{k-1} + \ldots + 2^0) \\
&\leq 2^{n-1} \cdot T(1) + c(2^{n-2} + \ldots + 2^0) \\
&\leq c \cdot 2^{n-1} + c2^n \\
&\leq c^* \cdot 2^n \\
&= O(2^n)
\end{aligned}$$

Complexity Analysis Example

```
int gcdIter(int a, int b){
  int minV = min(a,b);

  for(int gcd = minV; gcd >=1; gcd --){
    // upper bound to the lower bound
    if((a % gcd ==0) && (b % gcd ==0)) {
      return gcd;
    }
  }
  return 1;
}
```

Complexity Analysis

- Procedure for computing the running time:
  - Determine the bounds on the running times of the statements
  - Proceed up the program structure tree
  - Analyze compound statements only after their constituent parts have been analyzed
- Analysing programs is not always trivial

Example

- Euclid's Algorithm for computing the greatest common divisor.

```
int recursiveGCD(int a, int b) {
   if (b==0) return a;
   return gcd(b, a%b);
}
```

```
int gcd(int a, int b){

  while(b != 0){
    int reminder = a % b;
    a = b;
    b = reminder;
  }

  return a;
}
```

Try with 100 and 94
94 and 6
6 and 4
Try with 100 and 65
65 and 35
35 and 30

Example

```
int gcd(int a, int b){

  while(b != 0){
    int reminder = a % b;
    a = b;
    b = reminder;
  }

  return a;
}
```

- The number of iterations depends on the values of a and b.
- Values of a and b are monotonically decreasing.
- After 1 iteration we have a = min{a,b}.
- We can prove that after two iterations, the value of a is at most half of what it has been before.
  - Therefore, the complexity is O(log min{a,b}).

Theorem: Let a and b, a>=b, be inputs to gcd(int a, int b). Then after at most two iterations of the while loop we obtain a* where a* <= a/2.

Sketch of proof by case distinction:

- Value of a is monotonically decreasing and we always have a >=b.
- Assume that b> a/2. Then b'=a % b <=a/2 and a'=b holds in the next iteration and a* = a' % b'= b % b' <= a/2 after two iterations due to % operation.
- Assume b<=a/2. Then a*=b<=a/2 after 1 iteration.

Searching an array

- Array access is O(1)

- But if we search for an element in an array, what's the worst case? What's the best case?
- What are your assumptions?
- Do these assumptions matter?
- What's the big-O for searching an array, if we can't make any assumptions about its contents?

Searching an array

- If we know that the data is sorted then we can make assumptions about where the thing that we're searching for is.
- I have an integer array of unique integers 1,2,3,4,..,10, inserted into locations 0..9 in order.
- What can I say about all of the elements from location 0 to location 4?
- What if they weren't in order?

Binary Search

- In binary search we locate the middle element in our structure, or nearest to middle element, and look at it.
- Is it what we’re looking for? Stop.
- Is it less than what we’re looking for? Look at the elements larger than this one.
- Is it greater? Look at the smaller elements?
- Have we run out of elements? Stop!

Binary Search

```
bool binarySearch(int arr[], int obj, int start, int end){

  while (start <= end){
    int middle = (start+end)/2;
    if(arr[middle] == obj)
      return true;
    else if(arr[middle] > obj)
      end = middle-1;
    else
      start = middle +1;
  }

  return false
}
```

Binary Search

- Benefits:
  - We halve the search space each time. Locating the middle element in an array is an O(1) operation, so it doesn't add complexity.
  - We know if the element isn't there without having to search everything.
- What complexity is binary search?

Binary Search

- In binary search we:
  - halve the search space every time
  - don't have to search every element
- Intuitively, this is better than O(n). But what is it?
- We keep halving the search space - so it's better than O(n/2)... $O(\log_2 n)$ =$O(\log n)$,usually we drop the 2
- Remember logarithm rule to change basis fromb to c: $\mathbf{log}_c(n) = \mathbf{log}_c(b) * \mathbf{log}_b(n)$
- Example $\log_2(n) = \log_2(10)* \log_{10}(n)$
- This is for worst case! Average case is roughly the same.

Searching

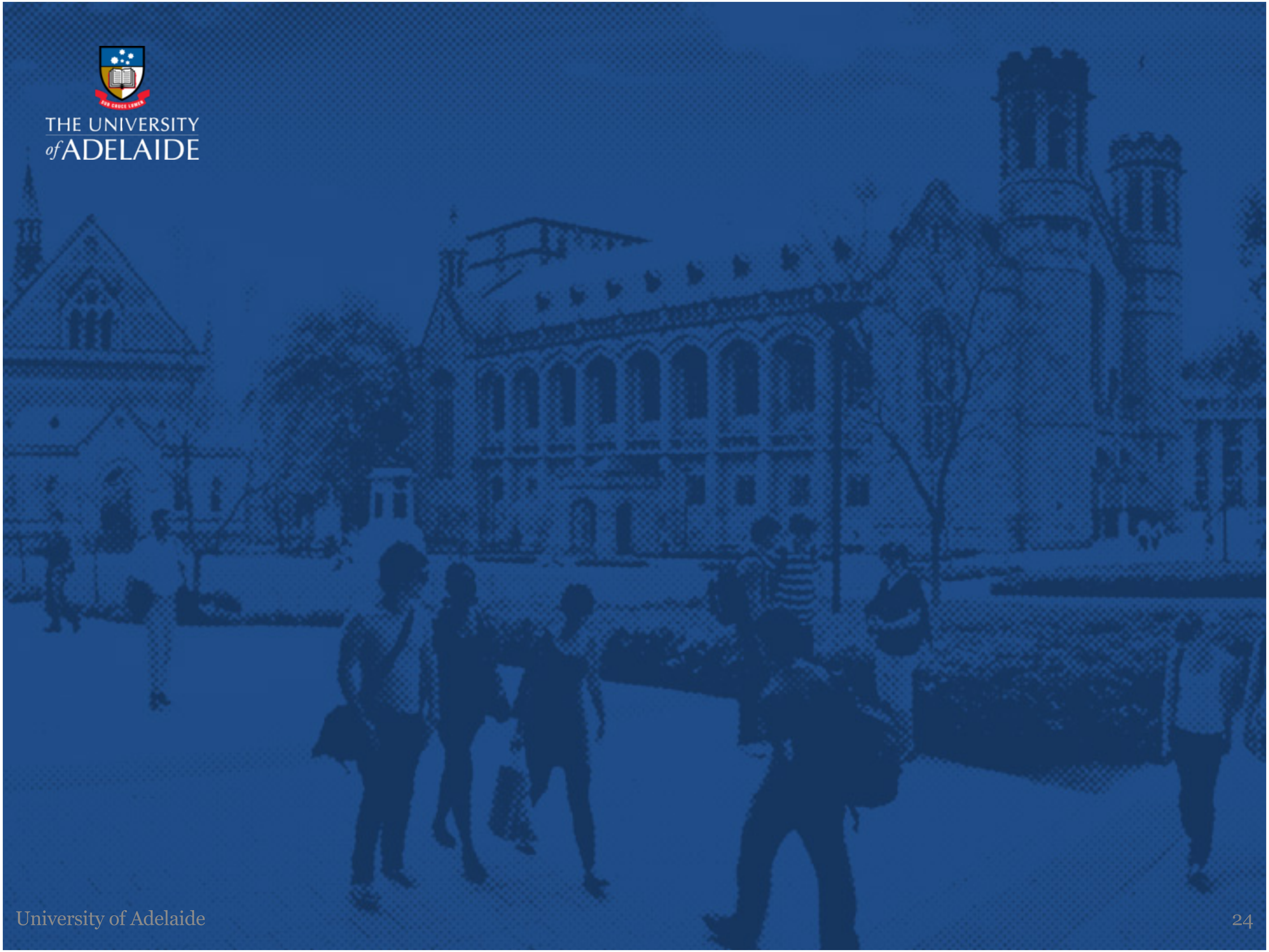
- Sorted data can be searched faster

- So if we can search sorted data in O(log n), this is a strong motivation to sort it in the first place.
- You've already seen selection sort and insertion sort in CS1102.
- What are their complexities?
- Why would we take the effort to sort, given that sorting effort is >= O(n)?

Sorting and Searching

- Sort once, search a lot

- We assume that, most of the time, we will search data far more frequently than we will sort it.
- Thus, a once-off sorting cost of $O(n^2)$ is acceptable, if we can then search at $O(\log n)$ thereafter.


THE UNIVERSITY
of ADELAIDE