Analysis: Worst Case

Class 31

Introduction

- in general, the amount of time a program takes to run is proportional to the amount of input it must process
- the more input, the more time it takes
- we care how much time a program takes to run because time is money

Introduction

- in general, the amount of time a program takes to run is proportional to the amount of input it must process
- the more input, the more time it takes
- we care how much time a program takes to run because time is money
- any program will run quickly with small input size
- so the real question is:

Scaling

How does the time taken by a program vary, or **scale**, as the amount of input grows?

Program vs Algorithm

- actually timing a program with a stopwatch has little value
- the number of seconds depends on
  - the language used
  - the speed and architecture of the computer it runs on
  - how heavily loaded the computer is with other jobs
- instead we wish to analyze the algorithm in a way that is
  - language-neutral
  - independent of hardware
  - independent of OS and load
- and so we analyze algorithms, a level of abstraction higher than programs

Time and Space

- we are interested both in time efficiency and space (RAM) efficiency
- often there is a trade-off between the two

- typically, we care more about time than space
- because you can buy more RAM, but you can't buy more time

Terminology

- let A be an algorithm
- let I be the input for A
- I has both size and arrangement
- we typically use n to denote the amount of input: $n = |I|$

- to analyze an algorithm, we count certain operations — more on this later

- assuming we know what operations we are counting, we define time(I) to be the number of operations executed by A, given I as input

Worst Case

- at your job, customers complain that an important page on the corporate website is too slow
- the company has done benchmarking and finds that the page never loads faster than 10 seconds but sometimes takes 45 seconds to load
- your boss gives you the assignment to rewrite the page
- you do so, and tell your boss you're done
- they ask what the performance now is, and you tell them it now sometimes loads in 5 seconds — a huge improvement over the old 10 seconds

Worst Case

- at your job, customers complain that an important page on the corporate website is too slow
- the company has done benchmarking and finds that the page never loads faster than 10 seconds but sometimes takes 45 seconds to load
- your boss gives you the assignment to rewrite the page
- you do so, and tell your boss you're done
- they ask what the performance now is, and you tell them it now sometimes loads in 5 seconds — a huge improvement over the old 10 seconds
- you tell your boss that it sometimes takes 90 seconds to load
- is your boss happy?

Worst Case

- both the best and worst case matter
  but the worst case is far more important
- we define the worst-case function

$$W_A(n) = \max(\text{time}(I))$$

  (note it would make more sense to use the notation $W_A(I)$)
- input I of size n is a worst-case input for A if, compared to all other inputs of the same size n, I causes A to execute the largest number of operations

Comparing Algorithms

- imagine A and B are two different algorithms
- they accomplish the exact same result, but differently (maybe A uses a for loop and B uses recursion)
- since they do the exact same thing, they accept the exact same input
- now, suppose that for any valid input I,

  $$W_A(I) \leq W_B(I)$$

  i.e., A always requires fewer or the same number of operations than B
- then A is more time-efficient than B

Optimal Algorithm

- if $W_A(I) \leq W_B(I)$ for every possible algorithm B (for every possible I), even for B's that have not yet been written, then A is optimal, in the worst case, for this task

- how do you demonstrate that A is better than any algorithm, even ones that have never been written or invented yet?
- the answer depends on finding a lower bound for the possible number of operations to compute the task

Proving Optimality

- to prove algorithm A is optimal for problem or task P, Hein gives a two-step process (page 289)
  1. find a function F for which $F(n) \leq W_B(n)$ for all positive n and for all algorithms B that solve P
  2. compare F and W_A and if $F = W_A$ then A is worst-case optimal

Find the Minimum: Unsorted

- given an unsorted list of elements a, find the minimum element

```
minimum = a.at(0);
for (size_t index = 1; index < a.size(); index++)
{
  minimum = a.at(index) < minimum ? a.at(index) : minimum;
}
```

- there are n elements in a
- the for loop body executes $n - 1$ times, each time executing one comparison
- the number of comparisons is thus $n - 1$, which is a lower bound

Binary Search

1	3	5	11	17	23	29

given a sorted list of values, a range of the list, and search key

1. if the range of elements is **empty**, return not-found sentinel

Binary Search

1	3	5	11	17	23	29

given a sorted list of values, a range of the list, and search key

1. if the range of elements is **empty**, return not-found sentinel
2. divide the range of elements to search into three regions:
    2.1 the very middle element

Binary Search

1	3	5	11	17	23	29

given a sorted list of values, a range of the list, and search key

1. if the range of elements is **empty**, return not-found sentinel
2. divide the range of elements to search into three regions:
   2.1 the very middle element
   2.2 the elements to the **left** of middle

Binary Search

1	3	5	11	17	23	29

given a sorted list of values, a range of the list, and search key

1. if the range of elements is **empty**, return not-found sentinel
2. divide the range of elements to search into three regions:
   - 2.1 the very middle element
   - 2.2 the elements to the **left** of middle
   - 2.3 the elements to the right of middle

Binary Search

1	3	5	11	17	23	29

given a sorted list of values, a range of the list, and search key

1. if the range of elements is **empty**, return not-found sentinel
2. divide the range of elements to search into three regions:
   - 2.1 the very middle element
   - 2.2 the elements to the **left** of middle
   - 2.3 the elements to the right of middle
3. if the search key value is the middle element, you've found it and you're done

Binary Search

1	3	5	11	17	23	29

given a sorted list of values, a range of the list, and search key

1. if the range of elements is **empty**, return not-found sentinel
2. divide the range of elements to search into three regions:
   2.1 the very middle element
   2.2 the elements to the **left** of middle
   2.3 the elements to the right of middle
3. if the search key value is the middle element, you've found it and you're done
4. else if the search key value is **smaller** than the middle element, repeat step 1 on the left half range

Binary Search

1	3	5	11	17	23	29

given a sorted list of values, a range of the list, and search key

1. if the range of elements is **empty**, return not-found sentinel
2. divide the range of elements to search into three regions:
   2.1 the very middle element
   2.2 the elements to the **left** of middle
   2.3 the elements to the right of middle
3. if the search key value is the middle element, you've found it and you're done
4. else if the search key value is **smaller** than the middle element, repeat step 1 on the left half range
5. else repeat step 1 on the right half range

Decision Trees

- most algorithms have if-then-else statements, also called branches
- the branch points can be depicted as nodes in a tree
- the leaves of the tree represent possible outcomes

- if the branches of an algorithm are strictly if-else statements, then the decision tree is a binary tree

- if the branches of an algorithm are if-else if-else statements, then the decision tree is ternary

Binary Search

- binary search has a ternary decision tree, because each question has three possible answers:
  1. we found the key at the middle of the range
  2. the key is in the left half of the range
  3. the key is in the right half of the range