Math Preliminaries:

- How mathematical: algebra, not calculus
  - will review
  - discrete math book, eg: Epp.
- Ceiling $\lceil\ \rceil$: $\lceil x \rceil$ = smallest int $\geq$ x, $\lceil 1.2 \rceil = 2$
- Floor $\lfloor\ \rfloor$: $\lfloor x \rfloor$ = biggest int $\leq$ x, $\lfloor 1.2 \rfloor = 1$
- $\sum$, $\prod$: $\sum$ = sum, $\prod$ = product
- factorial: $n! = 1*2*3*\ldots n$
  - $4! = 1*2*3*4 = 24$

Math Preliminaries:

- Exponential:
  - $a^{x+y} = a^x a^y$
  - $a^{xy} = (a^x)$^y
  - $a^{-x} = 1/a^x$
- Log: If $a^x = y$ then $\log_a y = x$.
  - $\log_2 8 = 3$, $\log_{10} 100 = 2$,$\log_2 1 = 0$
  - base 2, base e, e is Euler's number, ln
  - $\log xy = \log x + \log y$
  - $\log x^y = y \log x$

Algorithms – introduction

- What is an algorithm?
- Method of solving a problem .
- Instance: valid input .
- Terminate, unambiguous, 100% correct
- Difference between an algorithm and a program: algorithm at a higher level, but enough detail to implement as program.
- How to describe an algorithm: in English or pseudo code – **NOT CODE !**

Algorithms – introduction

- To see how to describe: look at examples in textbook or class.
  - Enough detail so reader can see what is going on, but not so much that loose sight of big picture.
- Problem : Find the largest number in an unsorted array
- In English: go through the array, keeping track of largest number seen so far as champion. If the current number is bigger than the champion, make the current number the champion.

Algorithms – introduction

- In pseudocode:

```
champ = A[1]
for i= 2 to n
    if A [i] > champ then
        champ = A[i]
Return (champ)
```

- In C/C++/Java code:
  - NOT OK

Measuring efficiency of Algorithms

- How to compare two diff. algorithms for efficiency i.e. how much time they take.
- Eg: searching in a sorted array.
- Linear search
- Binary search
- Which is faster ?
- Can linear search ever do better than binary search?
- Can binary search ever do better than linear search?
- So which is better ? How do we decide ?

Measuring efficiency of Algorithms

- Benchmarking/Run-time analyis: very useful, but dependent on machine, data, program, what other processes are operating at that time.
- Theoretical analysis: try to get an estimate of number of statements executed. How to do ?
- Average case: problems:
  - Averaged over what ?
  - Hard to do.
  - Makes no guarantee about one particular input

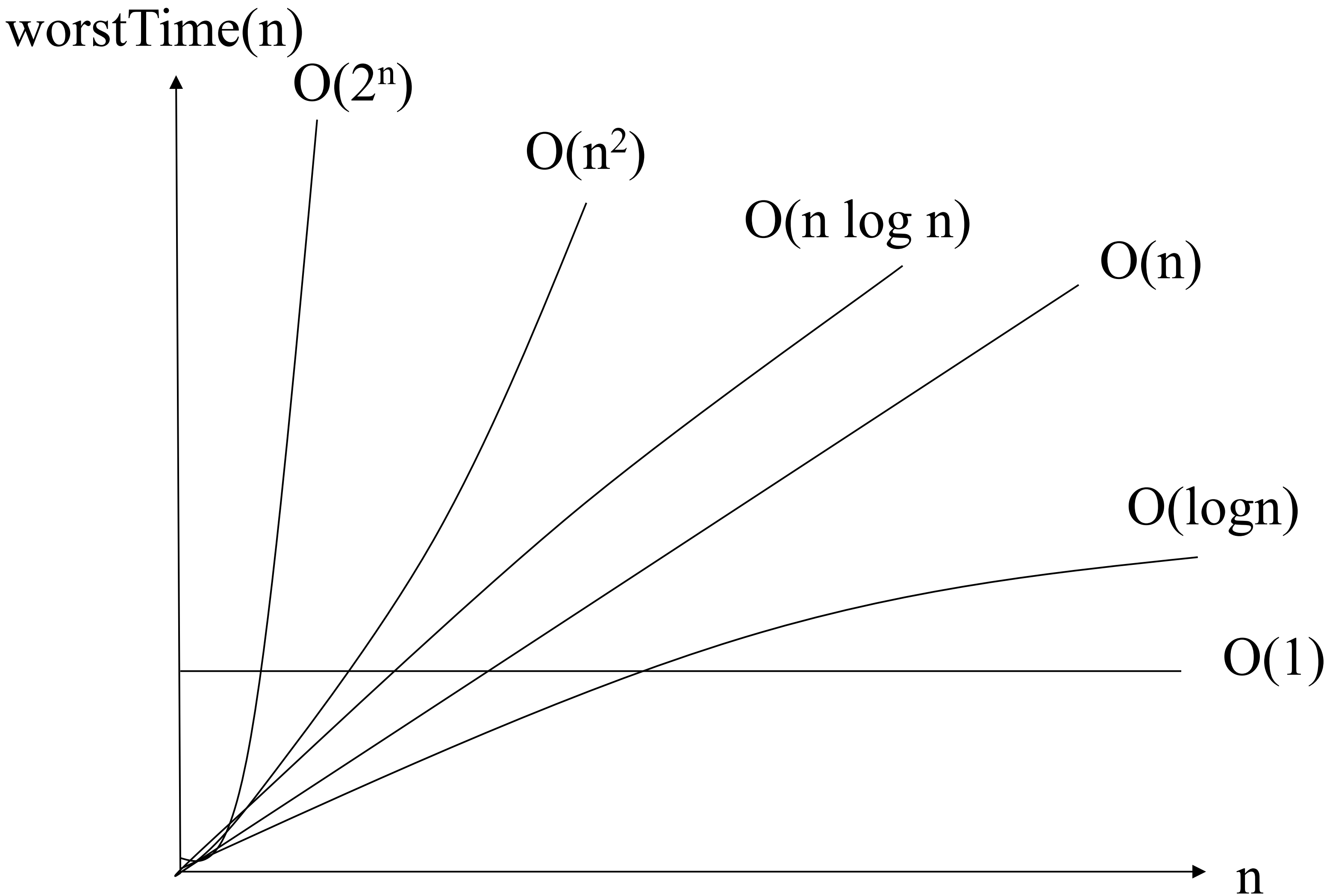
Worst case analysis

- How much time could the algorithm possibly take
  - Pessimistic but safe, easier to do than average case.
- How to compare: n better than n^2.
- Is 10n better than n^2 ?
  - For what n ?
  - We want to see what happens for larger values of n.
- Look at example

Big-Oh analysis

- Big-Oh notation: captures what happens for larger values of n ; asymptotic analysis.
- Definition: f(n) is O(g(n)) if there exists values M and t such that f(n) < M g(n) for x > t.
- Eg: 2n^2 is O(n^3), n^4 is not O(n^2)
- Graphical interpretation

Growth Rates



big-Oh properties

- x^a + x^(a-1) + x^(a-2) … is O(x^a)
- If a <= b, x^a is O(x^b)
- log x is O(x^a), x^a is O(2^x). Log functions < polynomial < exponential.
- log bases don't matter $\log_a$ x is O($\log_b$ x)
- If f(x) is O(h(x)), g(x) is O(h(x)) then
  f(x) + g(x) is O(h(x))
- If f(x) is O(h(x)), c * f(x) is O(h(x))

big-Oh properties/analysis

- $f(x)$ is $O(g(x))$ same as saying $f(x)$ belongs to $O(g(x))$.
- Best big-Oh calculation: 5x^2 is O(x^2), 5x^2 is O(x^3).
  - Better to write O(x^2) – why?
  - More accurate, think of algorithm analysis
- How to count # ops:
  - elementary ops count as 1
  - Do big-Oh analysis
  - Loops are crucial, nested loops

Examples

- sum = 0

  for (int j = 1; j <= n; j++)

  sum = sum + j
- for (int j = 0; j < n; j++)

  for (int k = 0; k < n; k++)

  println (j + k)
- Adding two matrices
- while (n > 1) n = n / 2;
- Linear search
- Binary search
- Now can we answer whether binary search is better than linear search ?

Common O( ) functions

- If worstTime(n) is ______, we will say
- "worstTime(n) is _______."
- O(1) … constant
- O(log n) … logarithmic in n
- O(n) … linear in n
- $O(n^2)$ … quadratic in n
- $O(2^n)$ … exponential in n

Sorting

- What is the sorting problem
- Input: Array A in arbitrary order
- Output: A in sorted order, smallest to biggest
- Why is it important i.e. why sort ?
- Makes searching faster. How ?
- Binary search
- How to sort: many different algorithms

Insertion sort

- Idea: At every point, elts on left sorted, elements on right unsorted. In single iteration:
  - Take next element from unsorted part
  - Insert it into sorted part
- Eg: 7 3 4 2 8 5

Procedure Insert (A[1..n])

```
for i ← 2 to n do {
/* put A[i] in correct spot */
   x ← T[i]; j ← i-1
   while (j > 0) AND (x< T[j])
         { T[j+1] ← T[j]
           j ← j-1}
    T[j+1] ← x
}
```

Insertion sort time analysis

- Worst case: analysis
- Can this be improved: i.e. could we do better ? To show not, find bad example.
- Best case: good input for insertion sort, time?
- Average case: intuition
- Ω Omega: for lower bounds
- Θ Theta: "same" if O() and Ω()

Merge sort

- Can find another sorting algorithm faster than insertion sort ?
  - Faster than $O(n^2)$
- Divide and conquer: will study later
- Merge algorithm:
  - Input: sorted arrays A,B
  - Output: combined sorted array C
  - How to do?
  - Eg: A= 2,4,5,8.B=1,3,9,10
- Time analysis of Merge:

Merge sort

- Input : array A with n elts
- Assume: n is power of 2
- merge sort (A)

  Break A into L,R size n/2

  merge sort (L)

  merge sort (R)

  merge (L,R) into A
- Eg: 7 2 3 5 4 9 1 6
- Time analysis: