CS310 - Advanced Data Structures and Algorithms

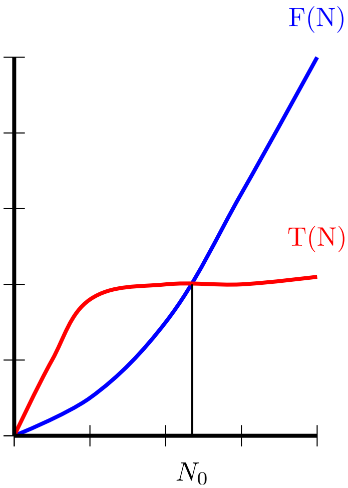
Spring 2014 – Class 2

January 30, 2014

Reading Material for this Class

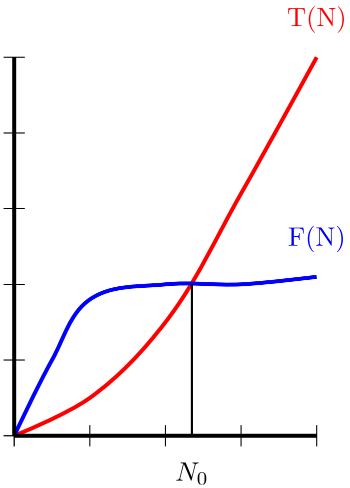
- Weiss chapter 5 (runtime analysis).
- Parts of chapter 7 (recursion).
- For next class read chapter 6, the part that talks about collections.

- $T(N)$ is $O(F(N))$ if there are positive constants c and N_0 such that $T(N) \leq c * F(N)$ for all $N \geq N_0$.
- $T(N)$ is bounded by a multiple of $F(N)$ from above for every big enough N.
- Example – Show that $2N + 4 = O(N)$



Runtime Analysis – Ω Notation

- $T(N)$ is $\Omega(F(N))$ if there are positive constants c and N_0 such that $T(N) \geq c * F(N)$ for all $N \geq N_0$.
- $T(N)$ is bounded by a multiple of $F(N)$ from below for every big enough N.
- Example – Show that $2N + 4 = \Omega(N)$



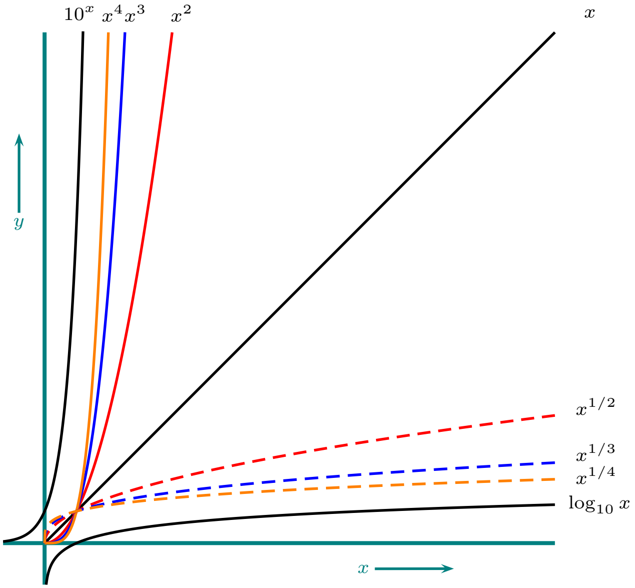
Runtime Analysis

- T and N are positive. N is the size of the problem and T measures a resource we want to measure: Runtime, CPU cycles, disk space, memory etc.
- Order of growth can be important. For example – sorting algorithms can perform quadratically or as $n * log(n)$. Very big difference for large inputs.
- We care less about constants, so $100N = O(N)$. $100N + 200 = O(N)$.
- Constant can be important when choosing between two similar run-time algorithm.
- Example – quicksort.

Runtime Analysis

- When the runtime is estimated as a polynomial we care about the leading term only.
- Thus $3n^3 + n^2 + 2n + 17 = O(n^3)$ because eventually the leading cubic term is bigger than the rest.
- For a good estimate on the runtime it's good to have both the O and the Ω estimates (upper and lower bounds).

Illustration



Adding and Multiplying Functions

- **Rule for sums** (e.g. - two consecutive blocks of code): If $T_1(N) = O(F(N))$ and $T_2(N) = O(G(N))$ then $T_1 + T_2 = O(\max(F(N), G(N))$. The biggest contribution dominates the sum.
- **Rule for products** (e.g. - an inner loop run by an outer loop): If $T_1(N) = O(F(N))$ and $T_2(N) = O(G(N))$ then $T_1 * T_2 = O(F(N) * G(N))$.
- **Example**:
  $(n^2 + 2n + 17) * (2n^2 + n + 17) = O(n^2 * n^2) = O(n^4)$.
  (Remember to ignore all but the leading term).
- If we sum over a large number of terms, we multiply the number of terms by the estimated size of one term.
- **Example**: Sum of i from 1 to N. Average size of an element: $\frac{N}{2}$. There are N terms so the sum is $O(N^2)$. Exact term: $\frac{N*(N-1)}{2}$.

Loops

- The runtime of a loop is the runtime of the statements in the loop * number of iterations.
- Example: bubble sort

```
/* sort array of ints in A[0] to A[n-1] */
int bubblesort(int A[], int n)
{
  int i, j, temp;
  for(i = 0; i < n-1; i++) /* n passes of loop */
    /* n-i passes of loop */
    for(j = n-1; j > i; j--)
      if (A[j-1] > A[j]) { // out of order:  swap
        temp = A[j-1]; A[j-1] = A[j]; A[j] = temp;
      }
}
```

Loops

- Work from inside out:
  - Calculate the body of inner loop (constant an if statement and three assignments).
  - Estimate the number of passes of the inner loop: n-i passes.
  - Estimate the number of passes of the outer loop: n passes. Each pass counts $n, n-1, n-2, \ldots, 1$.
  - Overall $1 + 2 + 3 + ... + n$ passes of constant operations: $\frac{n*(n-1)}{2} = O(n^2)$.
- This is not the fastest sorting algorithm but it's simple and works in-place. Good for small size input.
- We'll go back to sorting later in the course.

Recursive Functions

- Recursive functions perform some operations and then call themselves with a different (usually smaller) input.
- Example: factorial.

```
int factorial (int n)
{
  if(n<=1) return 1;
  return n*factorial(n-1);
}
```

Recursive Analysis

- Let us define T(n) as a function that measures the runtime.
- T(n) can be polynomial, logarithmic, exponential etc.
- T(n) may not be given explicitly in closed form, especially in recursive functions (which lend themselves easily to this kind of analysis).
- We have to find a way to derive the closed form from the recurrence formula.

Recursive Analysis

- Let us denote the run-time on input n as some function $T(n)$ and analyze $T(n)$.
- $O(1)$ operations before recursive call – if statement and a multiplication.
- The recursive part calls the same function with $n-1$ as input, so this part runs $T(n-1)$
- So: $T(n) = c + T(n-1)$.
- Similarly: $T(n-1) = c + T(n-2) \Rightarrow T(n) = 2c + T(n-2)$.
- After n such equations we reach $T(1) = k$ (just the if-statement).
- $T(n) = (n-1) * c + k = O(n)$.
- Iterative function performs the same.

A Problematic Example

The following function calculates 2^n for $n \geq 0$.

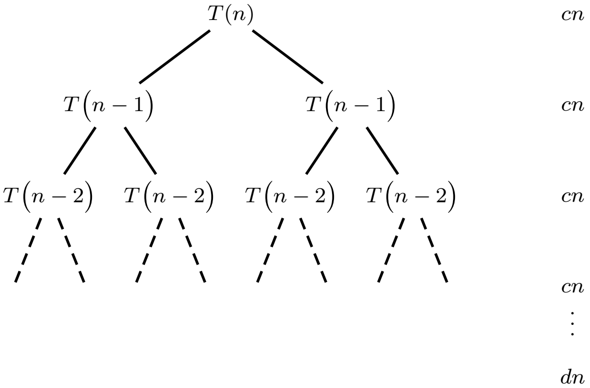
```
int power2(int n)
{
  if (n==0) return 1;
  return power2(n-1)+power2(n-1)
}
```

What is the problem here?

Ill-Behaved Recursion

- Each recursive call does a constant number of operations and spawns two recursive calls with n-1.
- $T(n) = c + 2 * T(n-1)$.
- $T(n-1) = c + 2 * T(n-2) \ldots T(2) = c + 2 * T(1)$.
- $T(1) = k$.
- C is positive and therefore:
- $T(2) > 2k$, $T(3) > 4k \ldots T(n) > 2^{n-1} * k$
- $T(n)$ is exponential with n!
- Intuitively, every call doubles the required solution time.

Ill-Behaved Recursion – Illustration



Ill-Behaved Recursion

- The problem is the double recursion which runs on the same input so we do a lot of redundant work.
- The call tree looks like a big binary tree.
- Double recursion is not bad, as long as we split the work too!
- Example: Merge sort – sort recursively two halves of an array and merge.
- Call recursively twice, but on different input! The work is split between recursive calls in a smart way.
- We make power2 more efficient by calling power2(n-1) only once and multiply the result by 2. What is the runtime now?

Logarithms

- Involved in many important runtime results: Sorting, binary search etc.
- Logarithms grow slowly, much more slowly than any polynomial but faster than a constant.
- Definition: $\log_B N = K$ if $B^K = N$. B is the base of the log.
- Examples:
  - $\log_2 8 = 3$ because $2^3 = 8$.
  - $\log_{10} 100 = 2$ because $10^2 = 100$.
  - $2^{10} = 1024$ (1K), so $\log_2 1024 = 10$.
  - 2^{20}=1M, so $\log 1M = 20$.
  - $2^{30} = 1$G so $\log 1G = 30$.

Logarithms

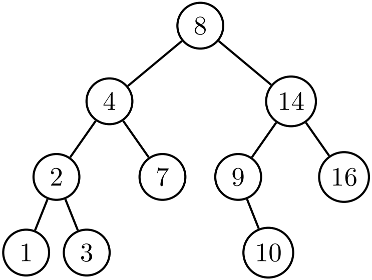
- It requires $\log_N K$ digits to represent K numbers in base N.
- It requires approx. $\log_2 K$ multiplications by 2 to get from 1 to N.
- It requires approx. $\log_2 K$ divisions by 2 to get from N to 1.
- Computers work in binary, so in order to calculate how many numbers a certain amount of memory can represent we use $\log_2$

Logarithms

- 16 bits of memory can represent 2^{16} different numbers $= 2^{10+6} = 2^{10} * 2^6 = 64K$.
- 32 bits of memory can represent 2^{32} different numbers $= 2^{30+2} = 2^{30} * 2^2 = 4G$ – see previous slide. (many of today's operating systems address space).
- 64 bits?? (most of today's computers address space).

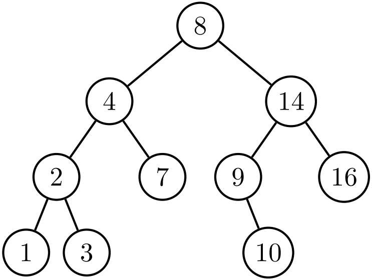
Binary Search Tree

- A very efficient way to hold data.
- The data is arranged in a binary tree structure so that every subtree rooted at element X holds:
- Left subtree elements always smaller than or equal to X.
- Right subtree elements always larger than X.



Binary Search Tree

- Searching the tree halves the search space at each stage.
- Searching the tree is logarithmic (why? Do analysis using T(N) as shown in class).
- Compare to linear search on a random array.



Recurrence Formula

$$T(n) = C \qquad \text{If n is 1}$$
$$T(n) = 2 * T(\tfrac{n}{2}) + cn \qquad \text{Otherwise}$$

Notice that c and C are not the same constant!

- Identities like this come up frequently in algorithmic analysis.
- It's important to have ways of solving them. We'll see a couple.
- One basic way is to form a recursion tree.

Recursion Tree

- If $N = 2^p$ then there are p rows with cn on the right, and one last row with dn on the right.
- Since $p = \log n$, this means that the total cost is $cN \log N + dN$ In other words, this is what we call an $O(N \log N)$ algorithm.