Growth of Functions

A way to describe behaviour of functions in the limit. We are studying asymptotic efficiency.

- Describe growth of functions.
- Focus on what’s important by abstracting away low-order terms and constant factors.
- How we indicate running times of algorithms.
- A way to compare “sizes” of functions:
  - O corresponds to \( \leq \)
  - \( \Omega \) corresponds to \( \geq \)
  - \( \Theta \) corresponds to =

We consider only functions \( f, g : \mathbb{N} \rightarrow \mathbb{R}^{\geq 0} \).

O-Notation

\( O(g(n)) \) is the set of all functions \( f(n) \) for which there are positive constants \( c \) and \( n_0 \) such that

\[
  f(n) \leq cg(n) \quad \text{for all } n \geq n_0.
\]

\( g(n) \) is an asymptotic upper bound for \( f(n) \).

If \( f(n) \in O(g(n)) \), we write \( f(n) = O(g(n)) \) (we will precisely explain this soon).

General remarks

- First we consider an important tool for the analysis of algorithms: Big-Oh.
- Then we introduce an important algorithmic paradigm: Divide-and-Conquer.
- We conclude by presenting and analysing two examples.

Reading from CLRS for week 2

- Chapter 2.3
- Chapter 3
\( 2n^2 = O(n^3), \) with \( c = 1 \) and \( n_0 = 2. \)

Example of functions in \( O(n^2) \):
- \( n^2 \)
- \( n^2 + n \)
- \( n^2 + 1000n \)
- \( 1000n^2 + 1000n \)

Also
- \( n \)
- \( n/1000 \)
- \( n^{1.999999} \)
- \( n^2 / \lg \lg \lg n \)

\( \Omega \)-Notation Examples
\( \sqrt{n} = \Omega(\lg n), \) with \( c = 1 \) and \( n_0 = 16. \)

Example of functions in \( \Omega(n^2) \):
- \( n^2 \)
- \( n^2 + n \)
- \( n^2 - n \)
- \( 1000n^2 + 1000n \)
- \( 1000n^2 - 1000n \)

Also
- \( n^3 \)
- \( n^{2.0000001} \)
- \( n^2 \lg \lg \lg n \)
- \( 2^n \)
Example 1

\[ n^2/2 - 2n = \Theta(n^2), \text{ with } c_1 = \frac{1}{4}, \ c_2 = \frac{1}{2}, \text{ and } n_0 = 8. \]

Theorem 2

\[ f(n) = \Theta(g(n)) \text{ if and only if } f(n) = O(g(n)) \text{ and } f(n) = \Omega(g(n)). \]

Leading constants and lower order terms do not matter.

Asymptotic notation chained together

\[ 2n^2 + 3n + 1 = 2n^2 + \Theta(n) = \Theta(n^2) \]

Interpretation:

- First equation: There exists \( f(n) \in \Theta(n) \) such that \( 2n^2 + 3n + 1 = 2n^2 + f(n) \).
- Second equation: For all \( g(n) \in \Theta(n) \) (such as the \( f(n) \) used to make the first equation hold), there exists \( h(n) \in \Theta(n^2) \) such that \( 2n^2 + g(n) = h(n) \).

Note

What has been said of \( \Theta \) on this and the previous slide also applies to \( O \) and \( \Omega \).

Asymptotic notation in equations

When on right-hand side

\( \Theta(n^2) \) stands for some anonymous function in the set \( \Theta(n^2) \).
\[ 2n^2 + 3n + 1 = 2n^2 + \Theta(n) \text{ means } 2n^2 + 3n + 1 = 2n^2 + f(n) \text{ for some } f(n) \in \Theta(n) \text{. In particular, } f(n) = 3n + 1. \]

When on left-hand side

No matter how the anonymous functions are chosen on the left-hand side, there is a way to choose the anonymous functions on the right-hand side to make the equation valid.
Interpret \( 2n^2 + \Theta(n) = \Theta(n^2) \) as meaning for all functions \( f(n) \in \Theta(n) \), there exists a function \( g(n) \in \Theta(n^2) \) such that \( 2n^2 + f(n) = g(n) \).

Example Analysis

**INSERTION-SORT**

1. \( \textbf{for } j = 2 \textbf{ to } A.length \)
2. \( \text{key} = A[j] \)
4. \( i = j-1 \)
5. \( \textbf{while } i > 0 \text{ and } A[i] > \text{key} \)
7. \( i = i-1 \)
8. \( A[i+1] = \text{key} \)

The for-loop on line 1 is executed \( O(n) \) times; and each statement costs constant time, except for the while-loop on lines 5-7 which costs \( O(n) \).
Thus overall runtime is: \( O(n) \times O(n) = O(n^2) \).

Note: In fact, as seen last week, worst-case runtime is \( \Theta(n^2) \).
Divide-and-Conquer Approach

There are many ways to design algorithms.

For example, insertion sort is incremental: having sorted \( A[1..j-1] \), place \( A[j] \) correctly, so that \( A[1..j] \) is sorted.

Divide-and-Conquer is another common approach:

- **Divide** the problem into a number of subproblems that are smaller instances of the same problem.
- **Conquer** the subproblems by solving them recursively.
  - *Base case:* If the subproblem are small enough, just solve them by brute force.
- **Combine** the subproblem solutions to give a solution to the original problem.

Divide-and-Conquer Min-Max

As we are dealing with subproblems, we state each subproblem as computing minimum and maximum of a subarray \( A[p..q] \). Initially, \( p = 1 \) and \( q = A.length \), but these values change as we recurse through subproblems.

To compute minimum and maximum of \( A[p..q] \):

- **Divide** by splitting into two subarrays \( A[p..r] \) and \( A[r+1..q] \), where \( r \) is the halfway point of \( A[p..q] \).
- **Conquer** by recursively computing minimum and maximum of the two subarrays \( A[p..r] \) and \( A[r+1..q] \).
- **Combine** by computing the overall minimum as the min of the two recursively computed minima, similar for the overall maximum.

### Naive Min-Max

Find minimum and maximum of a list \( A \) of \( n \geq 0 \) numbers.

**Naive-Min-Max(\( A \))**

1. \( least = A[1] \)
2. for \( i = 2 \) to \( A.length \)
   
   - if \( A[i] < least \)
     
     - \( least = A[i] \)
   
   - if \( A[i] > greatest \)
     
     - \( greatest = A[i] \)
3. return \((least, greatest)\)

The for-loop on line 2 makes \( n-1 \) comparisons, as does the for-loop on line 6, making a total of \( 2n-2 \) comparisons.

Can we do better? Yes!

**Divide-and-Conquer Min-Max Algorithm**

Initially called with \( \text{Min-Max}(A, 1, A.length) \).

**Min-Max(\( A, p, q \))**

1. if \( p = q \)
2. return \((A[p], A[q])\)
3. if \( p = q-1 \)
5. return \((A[p], A[q])\)
6. else return \((A[q], A[p])\)
7. \( r = \lfloor (p+q)/2 \rfloor \)
8. \((\text{min1, max1}) = \text{Min-Max}(A, p, r)\)
9. \((\text{min2, max2}) = \text{Min-Max}(A, r+1, q)\)
10. return \((\text{min(min1, min2), max(max1, max2)})\)

**Note**

- In line 7, \( r \) computes the halfway point of \( A[p..q] \).
- \( n = q - p + 1 \) is the number of elements from which we compute the min and max.
Solving the Min-Max Recurrence

Let \( T(n) \) be the number of comparisons made by Min-Max\((A, p, q)\), where \( n = q-p+1 \) is the number of elements from which we compute the min and max.

Then \( T(1) = 0 \), \( T(2) = 1 \), and for \( N > 2 \):

\[
T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + 2.
\]

Claim

\[
T(n) = \frac{3}{2}n - 2 \quad \text{for} \quad n = 2^k \geq 2, \ i.e., \ \text{powers of 2}.
\]

Proof.

The proof is by induction on \( k \) (using \( n = 2^k \)).

Base case: true for \( k=1 \), as \( T(2^1) = 1 = \frac{3}{2} \cdot 2^1 - 2 \).

Induction step: assuming \( T(2^k) = \frac{3}{2}2^k - 2 \), we get

\[
T(2^{k+1}) = 2T(2^k) + 2 = 2\left(\frac{3}{2}2^k - 2\right) + 2 = \frac{3}{2}2^{k+1} - 2 \quad \square
\]

Big-Oh, Omega, Theta by examples

- \( 5n + 111 = O(n) \) ? YES
- \( 5n + 111 = O(n^2) \) ? YES
- \( 5n + 111 = \Omega(n) \) ? YES
- \( 5n + 111 = \Omega(n^2) \) ? NO
- \( 5n + 111 = \Theta(n) \) ? YES
- \( 5n + 111 = \Theta(n^2) \) ? NO
- \( 2^n = O(3^n) \) ? YES
- \( 2^n = \Omega(3^n) \) ? NO
- \( 120n^2 + \sqrt{n} + 99n = O(n^2) \) ? YES
- \( 120n^2 + \sqrt{n} + 99n = \Theta(n^2) \) ? YES
- \( \sin(n) = O(1) \) ? YES

Solving the Min-Max Recurrence (cont’d)

Some remarks:

- If we replace line 7 of the algorithm by \( r = p+1 \), then the resulting runtime \( T'(n) \) satisfies \( T'(n) = \lceil \frac{3n}{2} \rceil - 2 \) for all \( n > 0 \).
- For example, \( T'(6) = 7 \) whereas \( T(6) = 8 \).
- It can be shown that at least \( \lceil \frac{3n}{2} \rceil - 2 \) comparisons are necessary in the worst case to find the maximum and minimum of \( n \) numbers for any comparison-based algorithm: this is thus a lower bound on the problem.
- Hence this (last) algorithm is provably optimal.

Unfolding the recursion for Min-Max

We have

\[
T(n) = \begin{cases} 
0 & \text{if } n = 1 \\
1 & \text{if } n = 2 \\
T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor) + 2 & \text{else} 
\end{cases}
\]

- \( T(1) = 0 \)
- \( T(2) = 1 \)
- \( T(3) = T(2) + T(1) + 2 = 1 + 0 + 2 = 3 \)
- \( T(4) = T(2) + T(2) + 2 = 1 + 1 + 2 = 4 \)
- \( T(5) = T(3) + T(2) + 2 = 3 + 1 + 2 = 6 \)
- \( T(6) = T(3) + T(3) + 2 = 3 + 3 + 2 = 8 \)
- \( T(7) = T(4) + T(3) + 2 = 4 + 3 + 2 = 9 \)
- \( T(8) = T(4) + T(4) + 2 = 4 + 4 + 2 = 10 \)
- \( T(9) = T(5) + T(4) + 2 = 6 + 4 + 2 = 12 \)
- \( T(10) = T(5) + T(5) + 2 = 6 + 6 + 2 = 14 \).

We count 4 steps +1 and 5 steps +2 — we guess \( T(n) \approx \frac{3}{2}n \).
Finding the best min-max algorithm

- As you can see in the section on the min-max problem, for some input sizes we can validate the guess $T(n) \approx \frac{3}{2} n$.
- One can now try to find a precise general formula for $T(n)$.
- However we see that we have $T(6) = 8$, while we can handle this case with 7 comparisons. So perhaps we can find a better algorithm?
- And that is the case:
  - If $n$ is even, find the min-max for the first two elements using 1 comparison; if $n$ is odd, find the min-max for the first element using 0 comparisons.
  - Now iteratively find the min-max of the next two elements using 1 comparison, and compute the new current min-max using 2 further comparisons. And so on ....

This yields an algorithm using precisely $\left\lceil \frac{3}{2} n \right\rceil - 2$ comparisons. And this is precisely optimal for all $n$.

We learn: Here divide-and-conquer provided a good stepping stone to find a really good algorithm.