Chapter 2. Algorithm (run-time) Analysis

- Run-time
- “Precise” run-time analysis
- Asymptotic run-time analysis
  - Asymptotic functions ($O$, $\Omega$, $\Theta$)
  - Guidelines on asymptotic run-time analysis:
    - Non-recursive algorithms
    - Recursive algorithms: recurrence relations

Run-time

- We need to have the following notions in place before discussing the run-time of an algorithm:
  - Problem size, e.g., number of elements in a file
  - Run-time metric, e.g., number of elements accessed, number of comparisons between elements.
- The analysis of run-time can be “precise” or asymptotic. Typically asymptotic is used.
- Note: truly precise analysis is impossible.
Run-time cases

- **Worst** case ‘guarantees’ that the run time won’t be longer than that.
- **Average** case ‘predicts’ the (probabilistically) expected run time.
- **Best** case ‘dreams’ about the shortest possible run time.

Precise run-time analysis: example

Algorithm Foo:
Input: integer x
array S[N] of nonnegative integers (all initialized to -1)
Output: integer
begin
if x already exists in S[i] (i ∈ [0, N-1])
then return the partial sum from S[0] to S[i].
else // x does not exist in S[i]
begin
if S is full
then shift all S[i]s to the left, insert x into S[N-1], and return N-1.
else // S is not full
then insert x into the first S[i] equal to -1 and return i.
end
end
A program code implementing the algorithm Foo

```c
function Foo(int N) {
    for (int i = 0; i < N; i++) {
        if (S[i] == x) { // already exists
            sum = 0;
            for (int j=0; j <= i; j++) sum += S[j];
            return sum;   // Case 1
        } else if (S[i] == -1) continue; // exit from the loop
    }
    // At this point, either i==N or S[i]==-1 holds.
    if (i == N) {  // array S[] is full
        for (j = 0; j < N-1; j++) S[j] = S[j+1];
        S[N-1] = x;
        return N-1;          // Case 2
    } else { // not full (S[i] == -1)
        S[i] = x;
        return i;               // Case 3
    }
}
```

Let's do some “precise” analysis of the running time.

Run-time
- Case 1:
  \[(a_0 + a_1 k) + a_2 k\] where \(0 \leq k \leq N\) and \(s[k] == x\)
- Case 2:
  \[(b_0 + b_1 N) + b_2 N\]
- Case 3:
  \[(c_0 + c_1 m)\] where \(0 \leq m \leq N\) and \(s[m] == -1\)

Best case: \(\min(a_0, c_0)\) -- constant
Worst case: \(\max(a_0+a_1 N, b_0+(b_1+b_2)N, c_0+c_1 N)\) -- linear
Average case: \(\text{avg}(a_0+a_1 N/2, b_0+(b_1+b_2)N, c_0+c_1 N/2)\) -- linear
Run-time vs. its growth rate

- Truly precise analysis of run-time is not feasible, nor meaningful.
  - How precise is precise enough?
  - Run-time varies significantly depending on many external factors such as compiler, programming language, and hardware implementation.
  - Run-time is not predictable due to caching.
- What’s more stable across external factors and over caching is the growth rate of run-time instead of the run-time itself.

Growth rate of a function

- Consider a function of N: NlogN, N, 2^N, N^2, and logN.
- Which one grows the fastest, that is, increases the fastest as N increases?
- The order of growth rate from the smallest first: logN < N < NlogN < N^2 < 2^N.
**Example: growth of N, NlogN, N^2, N^3**

<table>
<thead>
<tr>
<th>Input Size</th>
<th>Algorithm 1 (O(N^3))</th>
<th>Algorithm 2 (O(N^2))</th>
<th>Algorithm 3 (O(N \log N))</th>
<th>Algorithm 4 (O(N))</th>
</tr>
</thead>
<tbody>
<tr>
<td>N = 10</td>
<td>0.000009</td>
<td>0.000004</td>
<td>0.000006</td>
<td>0.000003</td>
</tr>
<tr>
<td>N = 100</td>
<td>0.002580</td>
<td>0.000109</td>
<td>0.000045</td>
<td>0.000006</td>
</tr>
<tr>
<td>N = 1,000</td>
<td>2.281013</td>
<td>0.010203</td>
<td>0.000485</td>
<td>0.000031</td>
</tr>
<tr>
<td>N = 10,000</td>
<td>NA</td>
<td>1.2329</td>
<td>0.005712</td>
<td>0.000317</td>
</tr>
<tr>
<td>N = 100,000</td>
<td>NA</td>
<td>135</td>
<td>0.064618</td>
<td>0.003206</td>
</tr>
</tbody>
</table>

*Figure 2.2 Running times of several algorithms for maximum subsequence sum (in seconds).*
Relative growth rates of functions

- Let $T(N)$ and $f(N)$ be functions of $N$.
- Let $\zeta(T(N))$ and $\zeta(f(N))$ be the growth rates of $T(N)$ and $f(N)$, respectively.
- Then, we say:
  - $T(N) = O(f(N))$ iff $\zeta(T(N)) \leq \zeta(f(N))$
  - $T(N) = \Omega(f(N))$ iff $\zeta(T(N)) \geq \zeta(f(N))$
  - $T(N) = \Theta(f(N))$ iff $\zeta(T(N)) = \zeta(f(N))$
  - $T(N) = o(f(N))$ iff $\zeta(T(N)) < \zeta(f(N))$

Asymptotic bounds of a function

- $T(N) = O(f(N))$ iff $\exists \ c_0 > 0, n_0 > 0 \ \forall \ N > n_0 \ T(N) \leq c_0 f(N)$ for $N > n_0$
- $T(N) = \Omega(f(N))$ iff $\exists \ c_0 > 0, n_0 > 0 \ \forall \ N > n_0 \ T(N) \geq c_0 f(N)$ for $N > n_0$
- $T(N) = \Theta(f(N))$ iff $\exists \ c_0 > 0, n_0 > 0 \ \forall \ N > n_0 \ T(N) = c_0 f(N)$ for $N > n_0$
  - i.e., iff $T(N) = O(f(N))$ and $T(N) = \Omega(f(N))$.
- $T(N) = o(f(N))$ iff $\exists \ c_0 > 0, n_0 > 0 \ \forall \ N > n_0 \ T(N) < c_0 f(N)$ for $N > n_0$
  - “there exists”
  - “such that”
Asymptotic bounds: example

\[ T(N) = 3 + 8N + 5N^2 = O(N^2) \]

**Proof:**
We need to show that
\[ \exists c_0 > 0, n_0 > 0 \ni T(N) \leq c_0 N^2 \text{ for } N > n_0. \quad (1) \]
Consider \( n_0 = 1 \), chosen arbitrarily.
Then, for all \( N > 1 \), \( T(N) = 3 + 8N + 5N^2 < c_0 N^2 \) for any \( c_0 > 16 \). Hence, there exist \( c_0 \) and \( n_0 \) that satisfy the condition (1).

Types of asymptotic bounds: \( O, \Omega, \theta \)

- \( O(f(N)) \): upper bound; worst case
- \( \Omega(f(N)) \): lower bound; best case
- \( \theta(f(N)) \): tight bound; worst case and best case are the same.
Illustrations of $O$, $\Omega$, $\Theta$

True or false?
1. $g_1(N)=O(f_1(N))$
2. $g_2(N)=O(f_1(N))$
3. $g_3(N)=O(f_1(N))$
4. $g_1(N)=\Omega(f_2(N))$
5. $g_2(N)=\Omega(f_2(N))$
6. $g_3(N)=\Omega(f_2(N))$

Big-Oh rules

- For any positive constant $c$, if $T(N) = O(c f(N))$ then $T(N) = O(f(N))$.
- If $T_1(N) = O(f_1(N))$ and $T_2(N) = O(f_2(N))$ then
  - $T_1(N) + T_2(N) = O(f_1(N) + f_2(N))$
  - $T_1(N)T_2(N) = O(f_1(N)f_2(N))$.
- If $T(N)$ is a polynomial of degree $k$, then $T(N) = \Theta(N^k)$.
- $\log^k N = O(N)$ for any constant $k$.
- This shows that logarithms grow much slower than linear.
- Exercise: plot $N, \log N, \log^2 N, \log^3 N, \ldots$
Side note: proof of $\log^k N = O(N)$

$\log^k N = O(N)$ is true if $\lim_{N \to \infty} (\log^k N / N) = 0$. ------- Eq.(1)

(Precisely speaking, $\log^k N = o(N)$, little-o of $N$, in this case.)

We prove this by induction.

For $k = 0$, $\lim_{N \to \infty} (\log^0 N / N) = \lim_{N \to \infty} (1/N) = 0$.

For $k = 1$, $\lim_{N \to \infty} (\log^1 N / N) = \lim_{N \to \infty} (\log N / N) = 0$.

Assume Eq.(1) is true for $k = i-1$, that is, $\lim_{N \to \infty} (\log^{i-1} N / N) = 0$.

Then, for $k = i$,

$\lim_{N \to \infty} (\log^i N / N) = \lim_{N \to \infty} (i \cdot \log^{i-1} N / N) = i \cdot \lim_{N \to \infty} (\log^{i-1} N / N) = 0$.

Q.E.D.

by L'Hopital’s rule:

$\lim_{N \to \infty} (f(N)/g(N)) = \lim_{N \to \infty} (f'(N)/g'(N))$ where $f'$ and $g'$ are the derivatives of $f$ and $g$ w.r.t. $N$, respectively.

Complexity of a problem

- The complexity of a problem refers to the worst case run-time of the best algorithm found for solving the problem.

- Example: NP-hard problems have no known algorithm with polynomial worst case run-time.
Asymptotic run-time of an algorithm

- Oftentimes asymptotic bounds are adequate enough to express the run-time complexity of an algorithm.

Basic rules of asymptotic run-time analysis

- Non-recursive function:
  - Fragment the code: loop, sequence, conditional branching
  - Analyze the run-time of each fragment
  - Combine the run-time analysis results.

- Recursive function:
  - Build a recurrence relation.
Loop

\[
\begin{align*}
\text{for } i &= 1 \text{ to } N \\
\text{for } j &= 1 \text{ to } M \\
\text{do something in } O(1); \\
\end{align*}
\]

\[O(MN)\]

Side note: We can say \(O(MN) = O(M^2)\) if there exist constants \(c_0, n_0, m_0\) such that \(N \leq c_0M\) for \(N > n_0, M > m_0\).

Sequence

\[\begin{cases}
O(f(N)) \\
O(g(N))
\end{cases} \rightarrow \begin{cases}
O(f(N)) + O(g(N)) \\
\text{// Max}(O(f(N)), O(g(N)))
\end{cases}\]
Conditional branching

If (condition)
   then do something in $O(f(N))$
   else do something else in $O(g(N))$

$\max(O(f(N)), O(g(N)))$

Analysis of a recursive algorithm

- Run-time analysis of a recursive algorithm involves building and solving a recurrence relation.
- A recurrence relation is a mathematical relationship expressing $f(n)$ as some combination of $f(i)$ with $i < n$.
  - A.k.a. recurrence equations or difference equations.
  - [http://mathworld.wolfram.com/RecurrenceRelation.html]
Example: binary search

// Search for x in an array a[i..j] sorted in the ascending order.
// If found then return the array index m such that a[m] = x
// else return NOT_FOUND;
int a[];
binsearch(x, i, j) {
    m = floor((i+j)/2);
    if (x == a[m]) then return m;
    else if (i == j) then return NOT_FOUND;
    else if (x < a[m]) then return binsearch(x, i, m-1);
    else return binsearch(x, m+1, j);
}

Run-time: the number of comparisons between array elements
Problem size: the number of elements in the array

Recurrence equation:
T(N) = T(N/2) + 2 if N >= 2
= 1 if N = 1.

Let N = 2^n. Then, recursively,
T(2^n) = T(2^(n-1)) + 2
T(2^(n-1)) = T(2^(n-2)) + 2
... 
T(2^2) = T(2) + 2
T(2) = T(1) + 2

------------------------{ +
T(2^n) = T(1) + 2n = 1 + 2n

Hence, T(N) = 1 + 2 logN = O(logN)

Side note: What we did here is to derive the asymptotic functional form of T(N). To be precise, T(N)=O(logN) would have to be proven (using the formal definition of big-Oh.)
Example: recursive minmax

\[
\text{minmax}(i, j) \{
    \text{if } (i==j) \text{ then return } <a[i], a[j]>; \\
    \text{else } \\
        <\text{min1}, \text{max1}> = \text{minmax}(i, \text{floor}(i+j)/2)); \\
        <\text{min2}, \text{max2}> = \text{minmax}(\text{floor}(i+j)/2+1, j)); \\
        \text{if (min1 > max2) then min1 = min2; } \\
        \text{if (max1 < max2) then max1 = max2; } \\
        \text{return } <\text{min1}, \text{max1}>; \\
    \}; \\
\};
\]

Example:

\[
\text{Minmax}(0,7) \\
\text{Minmax}(0,3) \text{ Minmax}(4,7) \\
\text{Minmax}(0,1) \text{ Minmax}(2,3) \text{ Minmax}(4,5) \text{ Minmax}(6,7) \\
\text{Minmax}(0,0) \text{ Minmax}(2,2) \text{ Minmax}(4,4) \text{ Minmax}(6,6) \\
\text{Minmax}(1,1) \text{ Minmax}(3,3) \text{ Minmax}(5,5) \text{ Minmax}(7,7) \\
\]

Array a[]: [ 1 | 8 | 3 | 12 | 3 | 9 | 11 | 8 ]
Run-time: the number of comparisons between array elements

Problem size: the number of elements in the array

Recurrence equation:
\[ T(N) = 2T(N/2) + 2 \text{ if } N \geq 2 \]
\[ = 0 \text{ if } N = 1. \]

Let \( N = 2^n \). Then, recursively,
\[ T(2^n) = 2 \cdot T(2^{n-1}) + 2 \]
\[ 2 \cdot T(2^{n-1}) = 2^2 \cdot T(2^{n-2}) + 2^2 \]
\[ 2^2 \cdot T(2^{n-2}) = 2^3 \cdot T(2^{n-3}) + 2^3 \]
\[ \ldots \]
\[ 2^{n-2} T(2^2) = 2^{n-1} T(2) + 2^{n-1} \]
\[ 2^{n-1} T(2) = 2^n T(1) + 2^n \]

\[ \text{------------------------------------------( +) } \]
\[ T(2^n) = 2^n T(1) + 2^n + 2^{n-1} + \ldots + 2^2 + 2 \]
\[ = 2^n + 2^{n-1} + \ldots + 2^2 + 2 \text{ (* because } T(1) = 0 \text{ *)} \]
\[ = 2(2^{n-1} + \ldots + 2 + 1) = 2^n - 2 \text{ (* geometric sum *)} \]
\[ = 2N - 2 = O(N) \]

Trivia question

Q. Both binsearch and minmax divides the array into two halves at each recursive call. Then, why is the run-time \( O(\log N) \) for binsearch whereas \( O(N) \) for minmax?

A. Because in the case of binsearch, the recursive binsearch is called only once for either of the two halves, whereas in the case of minmax it is called twice, once for each half.
Comparison with non-recursive minmax

```c
// Find the pair of <min, max> from an array.
minmax(i, j) {
    min = max = a[i];
    for k = (i+1) to j {
        if (a[i] < min) then min = a[i];
        if (a[i] > max) then max = a[i];
    }
};
```

Run-time = \( c_1 + c_2 \cdot 2^k \cdot (j-i+1) = O(N) \)
where \( N = j-i+1 \) (# of array elements), and \( c_1 \) and \( c_2 \) are constants.

Fibonacci number

- **Definition:**
  - \( \text{fib}(N) = \text{fib}(N-1) + \text{fib}(N-2) \) if \( N \geq 2 \)
  - \( = 1 \) if \( N = 0 \) or \( 1 \)
- **Fibonacci sequence:** 1, 1, 2, 3, 5, 8, 13, 21, 34,…

```c
int fib(int N) {
    if (N < 2) return 1;
    else return fib(N-1) + fib(N-2);
};
```
Run-time: the number of calls to the function Fib
Problem size: the input number N

Recurrence equation:
\[ T(N) = \begin{cases} T(N-1) + T(N-2) & \text{if } N \geq 2 \\ 0 & \text{if } N = 0 \text{ or } 1 \end{cases} \]

Trick: \( T(N) = \text{Fib}(N) \). They look the same!
Note that \( \text{Fib}(N) \approx \phi^N / \sqrt{5} \)
where \( \phi = (1 + \sqrt{5}) / 2 = 1.61803... \) ("golden ratio")
Hence, \( T(N) = O(\phi^N) \rightarrow \text{Exponential!} \)

Exercise: solve this recurrence equation in the same manner as the recursive binsearch and recursive minmax.