Measuring the Running Time of Programs

Fix a measure of the "size" $n$ of the data to which a program is being applied.

**Example:** For integer arguments, the value is often a good size measure. For strings: the length.

- Compute a big-oh upper bound on the running time of a program by induction on the complexity of program structures, i.e., the depth to which structures are nested.
  - Try to make the bound simple and tight.
- In the following, we assume there are no function calls in the program except for I/O operations.

**Basis:** *Simple statements* contain no statements nested within them. In C:

1. Assignment statements.
2. Goto’s, including `break`, `continue`, `return`.
3. Input/Output using function calls like `printf` or `getchar()`.

- Fundamental assumption: Application of an operator takes a constant amount of time.
  - Write "some constant" as $O(1)$.
  - Operators include arithmetic, comparison, logical.
  - ML has some exceptions: concatenation of strings or lists.
- Thus, in C every simple statement takes $O(1)$ time.

**Induction:** Complex statements built from simple statements by recursive application of:

1. Loop formers: `for-`, `while-`, `repeat-`.
2. Branching statements: `if-else-`, `if-`, `case-`.
3. Block formers: `{⋯}`.
Structure Trees

- node = complex statement; its children are the constituent statements.

Example:

```c
main() {
    int i;
    (1) scanf("%d", &i);
    (2) while(i>0) {
        (3) putchar('0' + i%2);
        (4) i /= 2;
    }
    (5) putchar('
');
}
```

Details of the Induction

- **Blocks**: Running time bound = sum of the bounds of the constituents.
  - Use summation law to drop from the sum any term that is big-oh of another term.

- **Conditionals**: Bound = $O(1) +$ larger of bounds for the if- and else- parts.
  - $O(1)$ is for cost of the test — usually negligible.

- **Loops**: Bound is usually the maximum number of times around the loop × the bound on the time to execute the loop body.
But we must include $O(1)$ for the increment and test each time around the loop.

The possibility that the loop is executed $0$ times must be considered. Then, $O(1)$ for the initialization and first test is the total cost.

Example: Consider binary-conversion function. Size of data = $i$.

- Lines 1, 3, 4, 5 each $O(1)$ by the basis.
- Block of 3–4 is $O(1) + O(1) = O(1)$.
- While of 2–4 iterates at most $\log_2 i$ times.
  Bound on body times number of iterations = $O(1) \times \log_2 i = O(\log i)$.
- Block of 1–5 is $O(1) + O(\log i) + O(1) = O(\log i)$.
- i.e., it takes $O(\log i)$ time to convert $i$ to binary by this function.

Triangular Double Loops

Sometimes we need to “give up” trying to tighten the upper bound on running time. Example: an inner loop iterates different numbers of times.

Example: Insertion sort: After $i$ iterations, the first $i$ elements of an array are sorted. At iteration $i + 1$ we move the $(i + 1)$st element forward until it meets an element smaller than it.

```c
void isort(int A[], int n) {
    int i, j;
    (1)  for(i=1; i<n; i++) {
    (2)    j = i;
    (3)    while(j>0 && A[j-1]>A[j]) {
    (4)        swap(j-1, j); /* exchange A[j-1] with A[j] */
    (5)        j--;
    (6)    }
    (7)  }
}```
• Input “size” = n = length of array A.

• Lines 2, 4, 5 are $O(1)$; note *swap* is short for 3 assignment statements.

• Block 4–5 is $O(1) + O(1) = O(1)$.

• While-loop 3–5 iterates at most i times, for $j = i$ down to $j = 1$.
  - Thus, $i \times O(1) = O(i)$ is an upper bound on lines 3–5.

• Block 2–5 takes time $O(1) + O(i) = O(i)$.

• For-loop 1–5 iterates $n - 1$ times. The body takes $O(i)$ time.
  - But $i$ changes within the loop and makes no sense outside the loop, so we cannot say the for-loop takes $O(ni)$ time.
  - But $n \geq i$, so $O(n)$ is an upper bound on the while-loop of 3–5.
  - Then, the upper bound on the for-loop is $n \times O(n) = O(n^2)$.

• Nothing lost. If we summed the times for each iteration of the for-loop we would get $\sum_{i=1}^{n-1} O(i) = O(n(n-1)/2) = O(n^2)$. 