# Notes, 3(c)

ECE 606

## Efficiency

Two axes: time and space.

## Time-efficiency

Usually characterized as a function of (1), or both (1) and (2):

1. Number of certain kinds of operations, e.g., comparisons, multiplications.
2. Size, i.e., # bits, of operands of those operations. Two kinds of encodings are of particular interest:
   - Binary = base-2. Turns out is faithful representative for any integer base $\geq 2$.
   - Unary = base-1. Will come up occasionally, later in the course.

Case-study: isInArray where every entry in the array, and the item $i$ that is searched for, is a positive integer.

```
isInArray($A[1, \ldots, n], i$)
1 foreach $j$ from 1 to $n$ do
2     if $i = A[j]$ then return true
3 return false
```

We could count:

(A) Number of assignments of $j$ in Line (1) and/or comparisons "$i = A[j]$" in Line (2).

- If we stop with (A), this would correspond to (1) only from the previous page.

(B) The time each such assignment and comparison takes.

Observations:

(1) Number of comparisons is a function of the input. Specifically:

- $n$
- Whether $i \in A[1, \ldots, n]$ and in what spot.

(2) The time each comparison takes is a function of the input. Specifically:

- The size, i.e., # bits, to encode $i$ and each $A[j]$.

(3) Time to assign values to $j$ is a function of the input. Specifically:

- $n$
- Number of iterations we perform of the **foreach** loop
- Whether we assign only the number of bits that correspond to the current value of $j$

Point (1) above addressed by considering various cases:

- Worst-case: elicited by $i \notin A[1, \ldots, n]$ or by $i = A[n]$.
- Best-case: elicited by $i = A[1]$.
- Average-/Expected-case: based on assumptions. E.g.,:
  - $i \in A[1, \ldots, n]$, and,
  - Each $A[j]$ is distinct, and,
  - $i$ equally likely to be any one of $A[1], \ldots, A[n]$.

Point (2) above addressed by simplifying assumptions:

- Assume $i$ and every $A[j]$ encoded by same size.
  - As $i, A[j]$ are positive integers, this size would be $\lfloor \log_2 i \rfloor + 1$.
    - E.g., to represent a positive integer $t$ encoded in base-10, we need $\lfloor \log_{10} t \rfloor + 1$ digits.
    - E.g., to encode 1245, we need $\lfloor \log_{10}(1245) \rfloor + 1 = 4$ digits.
  - Can relax this assumption, and just adopt the size of:
    - $\max\{i, A[1], \ldots, A[n]\}$ — conservative/worst-case.
    - $\min\{i, A[1], \ldots, A[n]\}$ — optimistic/best-case.
- Assume that every comparison requires comparing every bit of $i$ and $A[j]$ — conservative/worst-case.
  - E.g., to compare the base-10 numbers 2034 and 1999, we do not need to compare every digit starting at most significant digit.
  - E.g., to compare the base-10 numbers 1998 and 1999, we need to compare every digit starting at most significant digit.

Point (3) above addressed by assumption:

- Assume we allocate space for $j$ once only, in a run of the algorithm.
- And this size corresponds to the maximum value $j$ needs to contain.
- Every time we assign a value to $j$, we assign all bits of that size, perhaps with leading 0's.
  - Then, encoding and assigning a value to $j$ requires $\lfloor \log_2 n \rfloor + 1$ bits.

Now our results for running-time.

- Worst-case: $n(\lfloor \log_2 i \rfloor + 1 + \lfloor \log_2 n \rfloor + 1)$
- Best-case: $\lfloor \log_2 n \rfloor + 1$ One assignment of j, and one comparison
- Expected-case: $E[X] \times (\lfloor \log_2 i \rfloor + 1 + \lfloor \log_2 n \rfloor + 1)$
  - $X$ is random variable = number of times we iterate in **foreach** loop.

$$\begin{aligned} E[X] &= 1 \times \Pr\{i = A[1]\} + 2 \times \Pr\{i = A[2]\} + \ldots + n \times \Pr\{i = A[n]\} \\ &= 1 \times \frac{1}{n} + 2 \times \frac{1}{n} + \ldots + n \times \frac{1}{n} \\ &= \frac{1}{n}(1 + 2 + \ldots + n) \\ &= \frac{1}{n} \times \frac{n(n+1)}{2} = \frac{n+1}{2} \end{aligned}$$

If we choose to represent time-efficiency with # assignments/comparisons only:

- Worst-case: $n$
- Best-case: 1
- Expected-case: $\frac{n+1}{2}$

Space-efficiency

This is the space the algorithm allocates in addition to that needed to encode the input of the algorithm. That is, the algorithm is not debited, or charged, for the space that is needed to write the input down. Only any additional space it allocates.

For isInArray, the only such auxiliary space it allocates is that for the variable $j$. So isInArray's space-efficiency can be meaningfully characterized as $\lfloor \log_2 n \rfloor + 1$. Because the largest value $j$ needs to contain is the positive integer $n$.

Sub-routine calls

If an algorithm invokes another algorithm, i.e., makes a "sub-routine call," or even itself, i.e., makes a recursive call, we need to account for that in our determination of time- and space-efficiency.

```
A(x)
1 $i \leftarrow x + 1$
2 $B(x)$
3 print $i$
```

```
B(y)
4 $i \leftarrow x + 10$
5 $C(y, i - 5)$
6 print $i$
```

```
C(p, q)
7 print $p, q$
```

From the standpoint of time-efficiency, analyzing the above situation is not really different from analyzing an algorithm with no subroutine calls.

E.g., a "hot operation" may be addition, and we could count the number of additions, which is 2. Or the number of times we print, which is also 2. Or even the maximum number of outstanding sub-routine calls at any time, which is 3.

But from the standpoint of space-efficiency, we need to be more careful. We need to consider how sub-routine calls are effected in practice. And that is: using a *call-stack*.

A stack is a natural data structure to use because we want exactly the LIFO property that a stack provides.

Recursive calls

```
isInArrayRecursive($A[1,\ldots,n], j, i$)
1  if $j > n$ then return false
2  if $A[j] = i$ then return true
3  return isInArrayRecursive($A, j+1, i$)
```

The above version is presumably first invoked as isInArrayRecursive($A, 1, i$). The time-efficiency of this version can be characterized to be exactly the same as the iterative version: the worst-case number of comparisons in Line (2) is $n$.

But the space-efficiency is dramatically different. The worst-case depth of the call-stack is $n$, which is exponentially worse than the the $\approx \log n$ space-efficiency of the iterative version.

This is one of the reasons that modern compilers eliminate so-called "tail recursion," which is a recursive call as the very last step in an algorithm. There is a standard way to replace a tail-recursive call with iteration — exactly like in isInArray.

## Asymptotics — $\Theta, O, \Omega$

Now that we have a way of characterizing the time- and space-efficiency of algorithms, as a final step, we ask how we can meaningfully compare two algorithms, call them $A$ and $B$, from the standpoint of time- and/or space-efficiency.

That is, suppose we have algorithms $A$ and $B$ for a problem, each of which takes some input of size $n$. Algorithm $A$ runs in time $2n + 100$, while algorithm $B$ runs in time $n^2$. Which of $A$ or $B$ is more time-efficient?

Answer: it depends. On the size of the input, i.e., value of $n$. For all $n \leq 11$, algorithm $B$ is more time-efficient. For all $n \geq 12$, algorithm $A$ is more time-efficient.

To give an unqualified answer, the customary approach in the context of algorithms is based on asymptotics, i.e., how the functions compare with one another as $n \to \infty$. For this example, we observe:

$$\lim_{n \to \infty} \frac{2n + 100}{n^2} = 0$$

Therefore we deem that algorithm $A$ is strictly more time-efficient than algorithm $B$ because, based on the value of the above limit, $n^2$ is a kind of upper-bound which is not tight for $2n + 100$.

Why is this mindset of looking at $n \to \infty$ meaningful? The reason is that it captures what happens at scale, i.e., as $n$ gets large. To concretize this, suppose you have a computer on which you plan to run algorithms $A$ and $B$, and your time-budget is $t$. Now we ask: given the time-budget $t$, what is the maximum problem-size we can handle?

The answer for algorithm $A$ is $n_a = \frac{t-100}{2}$; for $B$ is $n_b = \sqrt{t}$.

Now suppose we spend money and double the speed of our computer. This is equivalent to a new time-budget of $2t$. Then, the new input-size $n'_a$ we can handle if you run algorithm $A$ is $\approx 2n_a$ if $t \gg 100$. For algorithm $B$, $n'_b \approx 1.4n_b$. Thus, the pay-off with algorithm $A$ is better.

The difference is even more dramatic if we compare algorithm $B$ with algorithm $C$ which takes time $2^n$ for input-size $n$. Then, $n_c = \log_2 t$, and $n'_c = 1 + \log_2 t$. Thus, the pay-off with algorithm $C$ is poor: an input of size one more only. Whereas with algorithm $B$, we get a multiplicative pay-off.

We now introduce three notions to compare functions which correspond to upper-bound, lower-bound and tight-bound.

We assume that the functions of interest to us all: (i) have domain $\mathbb{N}$, (ii) have codomain $\mathbb{R}^+$, and, (iii) are non-decreasing, i.e., $x \geq y \implies f(x) \geq f(y)$.

<u>Asymptotic upper-bounds, $O(\cdot), o(\cdot)$</u>

We say that $f(n) = O(g(n))$, i.e., $g(n)$ is an "asymptotic upper-bound" for $f(n)$ if:

$$\text{there exists constant } c \in \mathbb{R}_0^+ \text{ such that } \lim_{n\to\infty} \frac{f(n)}{g(n)} = c$$

We say that $f(n) = o(g(n))$, i.e., $g(n)$ is an "asymptotic upper-bound that is not tight" for $f(n)$ if:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

<u>Asymptotic lower-bounds, $\Omega(\cdot), \omega(\cdot)$</u>

We say that $f(n) = \Omega(g(n))$, i.e., $g(n)$ is an "asymptotic lower-bound" for $f(n)$ if:

$$\text{there exists constant } c \in \mathbb{R}_0^+ \text{ such that } \lim_{n\to\infty} \frac{g(n)}{f(n)} = c$$

We say that $f(n) = \omega(g(n))$, i.e., $g(n)$ is an "asymptotic lower-bound that is not tight" for $f(n)$ if:

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$$

<u>Asymptotic tight-bound, $\Theta(\cdot)$</u>

We say that $f(n) = \Theta(g(n))$, i.e., $g(n)$ is an "asymptotic tight-bound" for $f(n)$ if: $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

<u>Discrete versions</u> For each of the above, a discrete version of the definition is in the textbook. E.g., we say $f(n) = O(g(n))$ if: there exist positive constants $n_0, c$ such that for all $n > n_0$, $f(n) \leq c \cdot g(n)$.

Example: binary search

Input: (i) $A[1,\ldots,n]$ of sorted positive integers, (ii) $lo, hi$ with $1 \leq lo \leq hi \leq n$, and, (iii) a positive integer $i$.

Output: true if $i \in A[lo,\ldots,hi]$, false otherwise.

```
BinSearch($A, lo, hi, i$)
1 while $lo \leq hi$ do
2     $mid \leftarrow \left\lfloor \frac{lo+hi}{2} \right\rfloor$
3     if $A[mid] = i$ then return true
4     if $A[mid] < i$ then $lo \leftarrow mid + 1$
5     else $hi \leftarrow mid - 1$
6 return false
```

Efficiency

Space-efficiency: the only additional space is that for *mid*. As the largest that *mid* can be is $n$, the number of items in the array $A$, the space-efficiency can be characterized as $\lfloor \log_2 n \rfloor + 1$.

Time-efficiency: we argue that the number of iterations of the **while** loop is a meaningful measure of the time-efficiency of BinSearch. And we prove in the textbook that this is $\leq 1 + \log_2 (hi - lo + 1)$.

## Polynomial-time, exponential-time

We say that an algorithm runs in *polynomial-time* if its time-efficiency is $O(n^c)$ for some positive constant $c$, where $n$ is the size of the input.

- Linear-time: if $c = 1$.
- Sub-linear-time: if time-efficiency of algorithm is $o(n)$.

We say that an algorithm runs in *exponential-time* if its time-efficiency is $O\big(2^{p(n)}\big)$ where $n$ is the size of the input and $p(n)$ is some polynomial in $n$.

Tip: a good way to not make mistakes with regards to the term polynomial-time is to use its fuller form: "in time polynomial in the size of the input."