Jan 12, 2017

1 Algorithm in Computing

Definition. *An **algorithm** is any well-defined computational procedure that takes an input to an output*

Definition. ***Sorting Problem***
***Input**: A sequence of n numbers* $< a_1, a_2, \cdots, a_n >$
***Output**: A permutation (reordering)* $< a'_1, a'_2, \cdots, a'_n >$ *such that* $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

Definition. *An algorithm is said to be **correct** if, for every instance, it hals with the correct output. Here, **instance** of a problem consists of the input (satisfying constraints) needed to compute a solution to the problem. We say a correct algorithm **solves** the given computational problem.*

Definition. *A **data structure** is a way to store and organize data in order to facilitate access and modification*

2 Getting Started

Insertion Sort An efficient algorithm for sorting a small number of elements.

Remark. Like playing cards. Start with empty hands and cards face down on the table. We remove card one at a time from table and insert it to correct position in left hand. To find the correct position, we compare it with each of cards already in hand from right to left

Algorithm 1: INSERTION SORT

Input: An array of elements A unsorted
Output: Array A is sorted

```
1 for j = 2 to A.length do
2     key = A[j]          /* insert A[j] into the sorted sequence A[1..j − 1] */
3     i = j − 1
4     while i > 0 and A[i] > key do
5         A[i + 1] = A[i]
6         i = i − 1
7     A[i + 1] = key
```

Definition. ***Analyzing** an algorithm has come to mean predicting the resources that the algorithm requires*

Definition. *The **input size** depends on problem being studied.*

- *number of items in the input for sorting problems*

- *number of bits for multiplying two integers*
- *more than a single number for graphs*

Definition. *The **running time** $t(x)$ of an algorithm on a particular input x is the number of primitive operations or steps executed*

Remark. Basic assumption is that a constant time is required to execute each line of pseudocode. One line may take different amount of time than another and assume i line takes time c_i. However we can ignore the abstract cost c_i when computing running time

Note. The running time for insertion sort depends on which input given a fixed size. The best case occurs if the array is already sorted, where $T(n) \in \mathcal{O}(n)$. If the array is in reverse sorted order, the worst case results, the inner while loop is executed j times. The worst case runing time follows $T(n) \in \Theta(n^2)$

Definition. *The **worst-case running time** is the longest runing time for any input of size n.*

$$T(n) = max\{t(x) : x \text{ is an input of size } n\}$$

Remark. Finding the worst case running time gives us an upper bound on the running time of any input. Knowing it provides a gurantee that the algorithm will never take any longer. The average case is often roughtly as bad as the worst case. We are most interested in the **rate of growth** of running time by ignoring the leading coefficients and lower order terms.

Definition. ***Divide-and-conquer** is an approach that break the problem into several subproblems that are similar to the original problem but smaller in size, solve the subproblems recursively, and then combine these solutions to create a solution to the original problem*

Algorithm 2: MERGE

```
1  Function Merge (A, p, q, r)
       Input: A is an array and p, q, r are indices in the array such that p ≤ q < r. Also
              A[p..q] and A[q + 1..r] are sorted
       Output: A[p..r] retains the same element and sorted
2      n_1 = q − p + 1
3      n_2 = r − q
4      let L[1..n_1 + 1] and R[1..n_2 + 1] be new arrays
5      for i = 1 to n_1 do
6          L[i] = A[p + i − 1]
7      for j = 1 to n_2 do
8          R[j] = A[q + j]
9      L[n_1 + 1] = ∞                          /* Insert a sentinel ∞ to the end */
10     R[n_2 + 1] = ∞                  /* Avoids having to check if L, R are empty */
11     i = j = 1
12     for k = p to r do
13         if L[i] ≤ R[j] then
14             A[k] = L[i]
15             i = i + 1
16         else
17             A[k] = L[j]
18             j = j + 1
19
```

```
1  Function Merge-Sort (A, p, r)
2      if p < r then
3          q = ⌊(p + r)/2⌋
4          Merge-Sort (A, p, q)
5          Merge-Sort (A, q + 1, r)
6          MERGE(A, p, q, r)
```

Definition. ***Recurrence relation*** *describes overall running time on a problem of size n in terms of the running time on smaller inputs.*

$$T(n) = \begin{cases} \Theta(1) & n \leq c \\ aT(\frac{n}{b}) + D(n) + C(n) & \textit{otherwise} \end{cases} \qquad (D \text{ - divide; } C \text{ - combine})$$

Proposition. *A* ***recursion tree*** *with n leaves, i.e. input size, has* $\log_2(n) + 1$ *levels*

Growth of Functions

Definition. *The **asymptotic** efficiency of algorithms is how the running time of an algorithm increases with the size of the input in the limit, as the size of the input increases without bound. The **asymptotic notation** is a mathematical notation that describes the limiting behavior of a function when the argument tends towards a particular value or infinity.*

Definition.

$$\Theta(g(n)) = \{f(n) : \exists c_1, c_2, n_0 > 0 \text{ such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0\}$$

*where $g(n)$ is an **asymptotically tight bound** for $f(n)$*

Remark. The definition requires that every $f(n) \in \Theta(g(n))$ be **asymptotically nonnegative** (for sufficiently large n) Consequently the function $g(n)$ must be asymptotically nonnegative as well, otherwise $\Theta(g(n)) = \emptyset$. Also beware of conventions

- $f(n) = \Theta(g(n)) \iff n \in \Theta(g(n))$
- $2n^2 + 3n + 1 = 2n^2 + \Theta(n)$ is equivalent to $2n^2 + 3n + 1 = 2n^2 + f(n)$ such that $f(n) = \Theta(n)$

Proposition. *For any polynomial $p(n) = \sum_{i=0}^{d} a_i n^i$, where $a_i \in \mathbb{R}$ and $a_d > 0$ we have $p(n) = \Theta(n^d)$*

Definition.

$$\mathcal{O}(g(n)) = \{f(n) : \exists c, n_0 > 0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0\}$$

*where $g(n)$ is an **asymptotically upper bound** for $f(n)$*

Remark. The upper bound $\mathcal{O}(g(n))$ on the worst-case running time of an algorithm bounds every input. However a tight bound $\Theta(g(n))$ on the worst-case running time of an algorithm does not imply that it bounds every input. (i.e. Although for insertion sort, the worst case running time is $\Theta(n^2)$; the best-case input runs only in $\mathcal{O}(n)$, when the array is already sorted)

Definition.

$$\Omega(g(n)) = \{f(n) : \exists c, n_0 > 0 \text{ such that } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0\}$$

*where $g(n)$ is an **asymptotically lower bound** for $f(n)$*

Remark. We say that the running time of an algorithm is $\Omega(g(n))$ by no matter what particular input of size n (i.e. best / worst - case) is chosen for each value of n, the running time is bounded by $g(n)$. This is equivalent to $\Omega(g(n))$ for best-case running time.

Theorem. *For any two functions $f(n), g(n)$, we have $f(n) = \Theta(g(n))$ if and only if $f(n) = \mathcal{O}(g(n))$ and $f(n) = \Omega(g(n))$*

Definition.

$$o(g(n)) = \{f(n) : \forall c > 0 \exists n_0 > 0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0\}$$

*where $g(n)$ is an **upper bound that is not asymptotically tight***

Remark. For example, $2n = o(n^2)$ but $2n^2 \neq (n^2)$. Intuitively, $f(n)$ becomes insignificant relative to $g(n)$ as n approaches infinity, i.e. $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$

Definition. *Similarly, a **lower bound that is not asymptotically tight** may be defined as*

$$f(n) \in \omega(g(n)) \iff g(n) \in o(f(n))$$

Remark. For example $\frac{n^2}{2} = \omega(n)$. Intuitively $f(n)$ becomes arbitrarily large relative to $g(n)$ as n approaches infinity, i.e. $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$

Definition. ***Monotonicity*** *A functions is monotonically increasing if $m \leq n$ implies $f(m) \leq f(n)$. A function is strictly increasing if $m < n$ implies $f(m) < f(n)$*

Definition. *For any real number x, denote the greatest integer less than or eequal to x by $\lfloor x \rfloor$ and the least integer greater than or equal to x by $\lceil x \rceil$. Some important properties*

1. $x - 1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil < x + 1$
2. $\lceil n/2 \rceil + \lfloor n/2 \rfloor = n$

Definition. ***modular arithmetic*** *For any integer a and any positive integer n, the valule $a \mod n$ is the remainder of the quotient*

$$a \mod n = a - n\lfloor a/n \rfloor$$

so follows that

$$0 \leq a \mod n < n$$

Definition. *Given $d \geq 0$, a **polynomial in n of degree** d is a function $p(n)$ of the form*

$$p(n) = \sum_{i=0}^{d} a_i n^i$$

where a_i are coefifcients of the polynomial. A function is polynomial bounded if

$$f(n) = \mathcal{O}(n^k)$$