13 Growth Rates

How does the time to iterate through a recursive algorithm grow with the size of the input? We answer this question for two algorithms, one for searching and the other for sorting. In both case, we find the answer by solving a recurrence relation.

Binary Search. We begin by considering a familiar algorithm, binary search. Suppose we have a sorted array, $A[1..n]$, and we wish to find a particular item, x. Starting in the middle, we ask whether $x = A[(n+1)/2]$? If it is, we are done. If not, we have cut the problem in half. We give a more detailed description in pseudo-code.

```
l = 1; r = n;
while l ≤ r do m = (l + r)/2;
  if x = A[m] then print(m); exit
    elseif x < A[m] then r = m − 1;
    elseif x > A[m] then l = m + 1
  endif
endwhile.
```

Assuming $n = 2^k - 1$, there are $2^{k-1} - 1$ items to the left and to the right of the middle. Let $T(n)$ be the number of times we check whether $l \leq r$. We check once at the beginning, for $n = 2^k - 1$ items, and then some number of times for half the items. In total, we have

$$T(n) = \begin{cases} T(\frac{n-1}{2}) + 1 & \text{if } n \geq 2; \\ 1 & \text{if } n = 1. \end{cases}$$

In each iteration, k decreases by one and we get $T(n) = k+1$. Since $k = \log_2(n+1)$, this gives $T(n) = 1+\log_2 n$. We could verify this by induction.

A similar recurrence relation. Let us consider another example, without specific algorithm. Suppose we solve a problem of size n by first solving one problem of size $n/2$ and then doing n units of additional work. Assuming n is a power of 2, we get the following recurrence relation:

$$T(n) = \begin{cases} T(\frac{n}{2}) + n & \text{if } n \geq 2; \\ 0 & \text{if } n = 1. \end{cases} \quad (1)$$

Figure 11 visualizes the computation by drawing a node for each level of the recursion. Even though the sequence of nodes forms a path, we call this the *recursion tree* of the computation. The problem size decreases by a factor of two from one level to the next. After dividing $\log_2 n$ times, we arrive at size one. This implies that there are only $1 + \log_2 n$ levels. Similarly, the work at each level decreases by a factor of two from one level to the next. Assuming $n = 2^k$, we get

$$\begin{aligned} T(n) &= n + \frac{n}{2} + \ldots + 2 + 1 \\ &= 2^k + 2^{k-1} + \ldots + 2^1 + 2^0 \\ &= 2^{k+1} - 1. \end{aligned}$$

Hence, $T(n) = 2n - 1$.

level	#nodes	size		work per node	work per level
1	1	n		n	n
2	1	$n/2$		$n/2$	$n/2$
3	1	$n/4$		$n/4$	$n/4$
4	1	$n/8$		$n/8$	$n/8$

Figure 11: The recursion tree for the relation in Equation (1).

Merge Sort. Next, we consider the problem of sorting a list of n items. We assume the items are stored in unsorted order in an array $A[1..n]$. The list is sorted if it consists of only one item. If there are two or more items then we sort the first $n/2$ items and the last $n/2$ items and finally merge the two sorted lists. We provide the pseudo-code below. We call the function with $\ell = 1$ and $r = n$.

```
void MergeSort(ℓ, r)
  if ℓ < r then m = (ℓ + r)/2;
    MergeSort(ℓ, m);
    MergeSort(m + 1, r);
    Merge(ℓ, m, r)
  endif.
```

We merge the two sorted lists by scanning them from left to right, using n comparisons. It is convenient to relocate both lists from A to another array, B, and to add a so-called stopper after each sublist. These are items that are larger than all given items. In other words, we assume the two lists are stored in $B[\ell..m]$ and $B[m+2..r+1]$, with $B[m+1] = B[r+2] = \infty$. When we scan the two lists, we move the items back to A, one at a time.

```
void MERGE(ℓ, m, r)
  i = ℓ; j = m + 2;
  for k = ℓ to r do
    if B[i] < B[j] then A[k] = B[i]; i = i + 1;
                   else A[k] = B[j]; j = j + 1
    endif
  endfor.
```

Assume $n = 2^k$ so that the sublists are always of the same length. The total number of comparisons is then

$$T(n) = \begin{cases} 2T(\frac{n}{2}) + n & \text{if } n \geq 2; \\ 0 & \text{if } n = 1. \end{cases}$$

To analyze this recurrence, we look at its recursion tree.

Recursion Tree. We begin with a list of length n, from which we create two shorter lists of length $n/2$ each. After sorting the shorter lists recursively, we use n comparisons to merge them. In Figure 12, we show how much work is done on the first four levels of the recursion. In this example, there are n units of work per level, and $1 + \log_2 n$ levels in the tree. Thus, sorting with the merge sort algorithm takes $T(n) = n \log_2 n + n$ comparisons. You can verify this using Mathematical Induction.

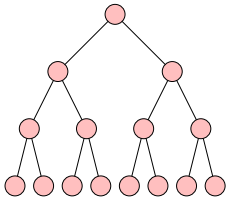
level	#nodes	size		work per node	work per level
1	1	n		n	n
2	2	$n/2$		$n/2$	n
3	4	$n/4$		$n/4$	n
4	8	$n/8$		$n/8$	n

Figure 12: The recursion tree for the merge sort algorithm.

Unifying the findings. We have seen several examples today, and we now generalize what we have found.

CLAIM. Let $a \geq 1$ be an integer and d a non-negative real number. Let $T(n)$ be defined for integers that are powers of 2 by

$$T(n) = \begin{cases} aT(\frac{n}{2}) + n & \text{if } n \geq 2; \\ d & \text{if } n = 1. \end{cases}$$

Then we have the following:

- $T(n) = \Theta(n)$ if $a < 2$;
- $T(n) = \Theta(n \log n)$ if $a = 2$;
- $T(n) = \Theta(n^{\log_2 a})$ if $a > 2$.

In the next lecture, we will generalize this result further so it includes our finding that binary search takes only a logarithmic number of comparisons. We will also see a justification of the three cases.

Summary. Today, we looked at growth rates. We saw that binary search grows logarithmically with respect to the input size, and merge sort grows at a rate of order $n \log_2 n$. We also discovered a pattern in a class recurrence relations.

23 Planar Graphs

Sixth Homework Assignment

Write the solution to each problem on a single page. The deadline for handing in solutions is 22 April 2009.