Mergesort and recurrences

Module 2: Algorithm Analysis

1 Overview

We introduce a new sorting algorithm called Mergesort and analyze its running time by writing and solving a recurrence relation. Recurrences are used to analyze recursive algorithms. We go over more examples of recurrences and how to solve them using repeated iteration.

Goals:

- Understand Mergesort: how it works, why it works, and its running time analysis using a recurrence
- Understand how to express the running time of recursive algorithms using recurrences
- Solve recurrences by repeated iteration
- Recognize broadly classes of recurrences (logarithmic, linear, $\Theta(n \lg n)$, exponential)

2 MergeSort

We saw a couple of $O(n^2)$ algorithms for sorting. Today we'll see a different approach that runs in $O(n \lg n)$ and uses an elegant technique called **divide-and-conquer** (we'll talk in more detail about this technique later in the semester). The high-level idea of Mergesort is the following:

Mergesort an array of n elements:

- Base-case: if size of input is 1, return
- Else:
  - Divide: Divide the array into two arrays of $n/2$ elements each
  - Conquer: Sort the two arrays recursively
  - Combine: Merge the two sorted arrays of $n/2$ elements into one sorted array of size n

- Note that it's a recursive algorithm. Each recursive call works on a smaller part of the original array.
- Let's delay for now the discussion of how to merge two sorted arrays, and focus on the recursion
- How to implement this idea? One way to write mergesort is to pass it the array to be sorted as argument, something like this:

```
mergesort (array A)
  – base case: if $A.size == 1$: return ($A$ is already sorted)
  – Else:
      ∗ Copy first half of A into A1 and second half of A into A2
      ∗ call recursively mergesort(A1) and mergesort(A2)
      ∗ merge A1 and A2 and write the output into A (now A is sorted)
```

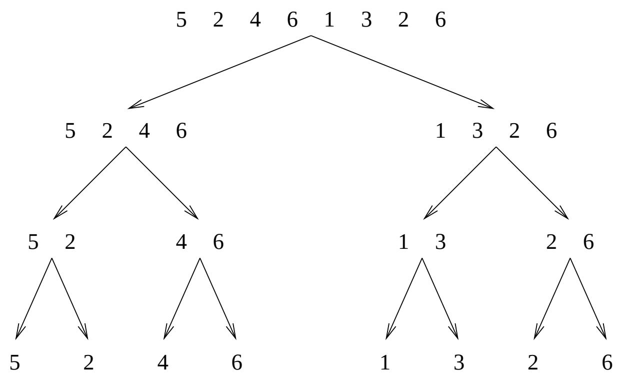
- Note that as written above the algorithm copies the array A into two sub-arrays $A1$ and $A2$. It is possible (and desirable) to avoid this copying: what we'll do instead is pass the original array A as argument along with two indices in the array, p and r, which will represent the part of the array that needs to be sorted.
- That is, we'll write a recursive procedure $MergeSort(A, p, r)$ which will sort the sub-array $A[p..r]$. When $MergeSort(A, p, r)$ returns, the sub-array $A[p..r]$ is assumed to be sorted.
- In order to sort the entire array we'll call $MergeSort(A, 0, n-1)$, that is, $p = 0$ and $r = n-1$.
- Similarly, the procedure to merge two sorted arrays will use indices to avoid copying: We will write a procedure Merge(A, p, q, r) which takes as arguments an array A and three indices p, q, r; when calling Merge(A, p, q, r) the arrays $A[p..q]$ and $A[q + 1....r]$ are both assumed to be sorted; the procedure will merge $A[p..q]$ and $A[q + 1....r]$ such that upon return $A[p..r]$ is sorted.
- We can write MergeSort of $A[p...r]$ as follows:

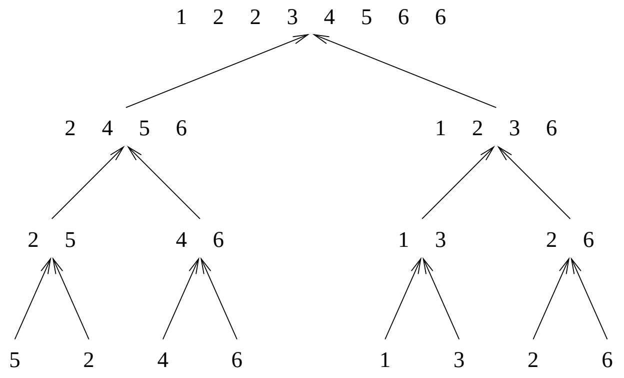
```
Merge Sort(A,p,r)
    If $p < r$ then
        $q = \lfloor (p + r)/2 \rfloor$
        MergeSort(A,p,q)
        MergeSort(A,q+1,r)
        Merge(A,p,q,r)
    //else: [nothing to do, array sizze < 2]
```

- How to implement **Merge(**A, p, q, r**)** ?
  - Idea: Imagine merging two sorted piles of cards. The idea is to choose the smallest of the two top cards and put it into the output pile. Then repeat.
  - We won't write the detailed implementation
  - **Analysis:** For each element that we put into the output pile, we compare the two elements at the "top" of the two piles. That's $O(1)$ work per element
  - Overall running time to merge $A[p..q]$ and $A[q + 1...r]$ is $\Theta(r - p)$

3 Mergesort Example

Consider calling Mergesort$(A, 0, 7)$ on an array of 8 elements. The first tree shows the recursive calls generated by this first call: two calls on arrays of 4 elements, which generate two calls each on two arrays of 2 elements, which in turn generate two calls on arrays of 1 element. The second tree shows how the sorted arrays produced by the recursive calls are merged together and returned to the caller.





4 Mergesort Correctness

- When we come up with an algorithm we always ask: is it correct? In our case: Does mergesort as written above correctly sort the array?
- Correctness proofs for recursive algorithms usually involve mathematical induction. Induction is not assumed/covered in this class, and if you are interested you can read up or take Math 2020 (Intro to proofs).
- Hand waving argument on why mergesort works: First, we claim that Merging two sorted arrays works because we always pick the smallest remaining element and put it next. Now for Mergesort: If the array has 1 element, then it's already sorted and Mergesort returns right away, so obviously it's correct. If the array has 2 elements, mergesort comes down to merging the two elements, and that works correctly. If the array has 4 elements, then mergesort makes

two recursive calls to sort arrays of 2 elements —- and we know those work correctly—-and then merges the results—and we said that merging works corectly. And so on. Basically we can show that if mergesorting any array of $n/2$ elements works correctly, then mergesorting array of n elements also works correctly.

5 Mergesort Analysis

- Now we want to analyze Mergesort on an array of n elements.
- To simplify things, let us assume that n is a power of 2, i.e $n = 2^k$ for some k.
- Let $T(n)$ denote the worst-case running time of mergesort on an array of n elements
- Running time of a recursive algorithm can be analyzed by writing a **recurrence relation**. Basically, think of $T(n)$ as a function of n and we'll express it in terms of itself.
- First note that if the array has size 1, mergesort returns in constant time. This says that $T(1) = \Theta(1)$.
- If $n > 1$, the time to mergesort an array of n elements consists of the time to sort recursively two arrays of size $n/2$, plus the time taken by merge
- Recursive calls: If the time to mergesort an array of n elements is $T(n)$, then the time to mergesort an array of $n/2$ elements is $T(n/2)$
- Merging: We know that the time to merge two arrays of size $n/2$ each is $\Theta(n)$
- Putting it all together, we have:

$$T(n) = \begin{cases} \Theta(1) & \text{If } n = 1 \\ 2T(n/2) + \Theta(n) & \text{If } n > 1 \end{cases}$$

- Great, we have a recurrence for $T(n)$! Now we need to solve it!
- Note that the recurrence for $T(n)$ is valid for any n, therefore is also valid for $n/2$ and for $n/4$, and so on. We can write $T(n/2) = 2T(n/4) + n/2$ and substituting this in the formula for $T(n)$, we get:

$$\begin{aligned} T(n) &= 2T(n/2) + n \\ &= 2(2T(n/4) + n/2)) + n \\ &= 4T(n/4) + 2n \end{aligned}$$

- Now we can do it again: $T(n/4) = 2T(n/8) + n/4$ and we get:

$$\begin{aligned} T(n) &= 4T(n/4) + 2n \\ &= 4(2T(n/4) + n/4) + 2n \\ &= 8T(n/8) + 3n \end{aligned}$$

And so on. We start seeing the pattern.

- General formula: After doing this i times we'll get:

$$T(n) = 2^i T(n/2^i) + i \cdot n$$

- Recursion depth: How long can we keep doing this? We can keep recursing as long as the problem size is > 1. When the problem size is 1 we reached the base case and we need to stop. The base case happens when

$$\frac{n}{2^i} = 1$$

or

$$i = \lg n$$

- Now we plug in the value for i in the general formula above and we get:

$$\begin{aligned} T(n) &= 2^{\lg n} T(1) + \lg n \cdot n \\ &= n \cdot T(1) + n \lg n \\ &= \Theta(n \lg n) \end{aligned}$$

- Thus we conclude that the worst-case running time $T(n)$ of Mergesort is $\Theta(n \lg n)$. Hooray!!

Independent work/reflection: Consider the best-case running time of Mergesort. What sort of input triggers it and what is its $\Theta()$?

6 Some final Mergesort notes

- Mergesort is the first sorting algorithm that we've seen which beats the quadratic bound of Bubblesort, Insertionsort and Selection sort.
- A drawback of Mergesort is that it is not *in place*.
- A sorting algorithm is called *in place* if it uses $O(1)$ space in addition to the input array. Bubblesort, Insertion sort and Selection sort are all in place. Mergesort, as written in this notes, is not, and the culprit for that is the merge procedure: it needs to create a new array, it cannot overwrite the input array. Think about it for a couple of minute to see why.

- Coming up with a Mergesort algorithm that's in place was an open problem for a while. An in-place Mergesort was eventually proposed by a researcher, but the algorithm is very complicated and is considered to be only of theoretical interest. In other words, Mergesort is not in place for all practical situations.
- Another drawback of Mergesort is that it's best case running time is $\Theta(n \lg n)$ (it would be nice if the best-case was $\Theta(n)$).
- When coming up with the recurrence relation for the running time we assumed, for simplicity, that n is a power of 2. If that's not the case, the recurrence is a little more complicated:

  $$T(n) = \begin{cases} \Theta(1) & \text{If } n = 1 \\ T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor) + \Theta(n) & \text{If } n > 1 \end{cases}$$

  It can be shown that the solution for this general case is the same. Basically what this means is that when solving recurrences it is ok to implicitly make simplifying assumptions as necessary (such as ignoring floors and ceilings, assuming n is a power of 2, etc) —these assumptions do not change the $\Theta()$ of the solution.

7 Solving recurrences by iteration: More examples

A recurrence relation is a recursive expression for the running time $T(n)$. Solving a recurrence means finding a theta-bound for $T(n)$. We will do this by *iteration*.[1]

7.1 Recurrence Example 2

- Problem: Find a $\Theta()$ bound for $T(n)$, where $T(n) = \begin{cases} \Theta(1) & \text{If } n = 1 \\ 3T(n/3) + n & \text{If } n > 1 \end{cases}$

- **Answer:**

  We know that:

  $T(n) = 3T(n/3) + n$

  and

  $T(n/3) = 3T(n/3/3) + n/3$

  and

  $T(n/9) = 3T(n/9/3) + n/9$

  and so on. We get:

$$\begin{aligned} T(n) &= 3T(n/3) + n \\ &= 3(3T(n/9) + n/3)) + n \\ &= 9T(n/9) + 2n \\ &= 9(3T(n/27 + n/9) + 2n \\ &= 27T(n/27) + 3n \\ &= \ldots \end{aligned}$$

- General formula:
$$T(n) = 3^i T(n/3^i) + i \cdot n$$

- Recursion depth: How long (how many iterations) it takes until the subproblem has constant size? i iterations, where $\frac{n}{3^i} = 1 \Rightarrow i = \log_3 n$

- Plug back in the formula

$$\begin{aligned} T(n) &= 3^i T(1) + i \cdot n \\ &= 3^{\log_3 n} T(1) + \log_3 n \cdot n \\ &= n \cdot T(1) + \log_3 n \cdot n \\ &= \Theta(n) + \Theta(n \log_3 n) \\ &= \Theta(n \lg n) \end{aligned}$$

[1]Note: Formally, after iteration finds a $\Theta()$ bound, it is necessary to use induction to prove the it formally. We'll skip this step.

7.2 Recurrence Example 3

- Problem: Find a $\Theta()$ bound for $T(n)$, where $T(n) = 8T(n/2) + n^2$ (with $T(1) = 1$)
- Answer: We know that:
  $T(n) = 8T(n/2) + n^2$
  and
  $T(n/2) = 8T(n/4) + (n/2)^2$
  and
  $T(n/4) = 8T(n/8) + (n/4)^2$
  and so on. We get:

$$\begin{aligned}
T(n) &= n^2 + 8T(n/2) \\
&= n^2 + 8(8T(\frac{n}{2^2}) + (\frac{n}{2})^2) \\
&= n^2 + 8^2T(\frac{n}{2^2}) + 8(\frac{n^2}{4})) \\
&= n^2 + 2n^2 + 8^2T(\frac{n}{2^2}) \\
&= n^2 + 2n^2 + 8^2(8T(\frac{n}{2^3}) + (\frac{n}{2^2})^2) \\
&= n^2 + 2n^2 + 8^3T(\frac{n}{2^3}) + 8^2(\frac{n^2}{4^2})) \\
&= n^2 + 2n^2 + 2^2n^2 + 8^3T(\frac{n}{2^3}) \\
&= \dots \\
&= n^2 + 2n^2 + 2^2n^2 + 2^3n^2 + 2^4n^2 + \dots
\end{aligned}$$

- Recursion depth: How long (how many iterations) until reach base case ? $\frac{n}{2^i} = 1 \Rightarrow i = \lg n$
- Plug back in:

$$\begin{aligned}
T(n) &= n^2 + 2n^2 + 2^2n^2 + 2^3n^2 + 2^4n^2 + \dots + 2^{\log n - 1}n^2 + 8^{\log n}T(1) \\
&= \sum_{k=0}^{\log n - 1} 2^k n^2 + 8^{\log n}T(1) \\
&= n^2 \sum_{k=0}^{\log n - 1} 2^k + (2^3)^{\log n}
\end{aligned}$$

- Now $\sum_{k=0}^{\log n-1} 2^k$ is a geometric sum so we have $\sum_{k=0}^{\log n-1} 2^k = \Theta(2^{\log n - 1}) = \Theta(n)$
- $(2^3)^{\log n} = (2^{\log n})^3 = n^3$
- And we get:

$$\begin{aligned}
T(n) &= n^2 \cdot \Theta(n) + n^3 \\
&= \Theta(n^3)
\end{aligned}$$

8 Recurrences: summary and final notes

Knowing recurrences and having an intuition of what common recurrences solve to can give you powerful hints of how to attack a problem. For example, if you were trying to solve a certain problem in linear time, it would be useful to know what sort of recurrences solve to linear time.

Some important/frequent recurrences:

- Logarithmic: $\Theta(\log n)$
  - Recurrence: $T(n) = T(n/2) + \Theta(1)$
  - Typical example: Recurse on half the input (and throw half away)
  - Variations: $T(n) = T(n/3) + \Theta(1)$, $T(n) = T(9n/10) + \Theta(1)$ (spend constant time and recurse on a fraction of the input)
- Linear: $\Theta(N)$
  - Recurrence: $T(n) = T(n-1) + 1$ (spend constant time and recurse on an input that's smaller by one element)
  - Also: $T(n) = 2T(n/2) + \Theta(1)$ (spend constant time and recurse on two halves)
  - Also: $T(n) = T(n/2) + \Theta(n)$ (spend linear time and recurse on half the input)
  - Varriations: $T(n) = T(2n/3) + \Theta(n)$ (spend linear time and recurse on a fraction of the input)
- Quadratic: $\Theta(n^2)$
  - Recurrence: $T(n) = n + T(n-1)$
  - Typical example: spend linear time and recurse on an input that's smaller by one element
- Exponential: $\Theta(2^n)$
  - Recurrence: $T(n) = 2T(n-1)$

Solving recurrences takes some practice, but after you have done a few, you will realize the process is the same and it's a matter of being careful. Hang in there!

We are done with the "discrete math" part of the class. We covered all tools we'll need for analysis (asymptotic notation, summations and recurrences), and from now on we'll focus on algorithms.