# Notes on Algorithms

Jennifer Zajac

Summer 2 2021

# Contents

# List of Tips

# Chapter 1

# Algorithms

## 1.1 Overview

**algorithm** An explicit, precise, unambiguous, mechanically-executable sequence of elementary instructions for solving a computational problem.

**algorithms** The study of how to solve computational problems.

**computational problem** Can be solved by computer; has a precise, well-defined, correct answer.

# Part I

# Problems

# Chapter 2

# Counting

Counting function

**Input:** A collection of objects.
**Problem:** Determine how many objects are in the collection.
**Output:** The number of objects (positive integer).

## 2.1 Simple Counting

Simple Counting with Students

```
1: Start count at 0 and pass ball to a student
2: repeat
3:     Student with ball sets their number to (1 + what last person said)
4:     Student with ball says their number Student with ball passes to a standing student & sits
5: until only 1 student is standing
6:
7: Student with ball sets their number to (1 + what last person said)
8: Student with ball says their number
```

**Explanation:** Each student counts for 1. Loop invariant: after each iteration, sum of standing students' numbers = total number of students. Every student is always accounted for.

**loop invariant** After every iteration of a loop, this property is true.

## 2.2 Recursive Counting

Recursive Counting with Students

```
Everyone set your number to 1
repeat
    Everyone partner up, wait if none found
    Set your number to (your number + partner's number), if waiting keep same number

    One partner sits down
until only 1 student is standing

The standing student says "the total is (their number)"
```

**Observations:** when there are $n$ students, and each line counts as 1 step...

- $T(1) = 2$.
- If $n > 1$, after 1 loop there are $\lceil n/2 \rceil$ students left. The ceiling considers the case of an odd student.
- $T(n) = 3 + T(\lceil n/2 \rceil)$, where there are 3 steps this loop and $T(\lceil n/2 \rceil)$ steps total in the following loops.
- $T(2^m) = 3m + 2$. Students get halved $m$ times.
  **Derivation:**

$$\begin{aligned}
&3 + T(\lceil 2^m/2 \rceil) && \text{Plugging in } 2^m.\\
&= 3 + T(\lceil 2^{m-1} \rceil) && 2^m \text{ is an integer. Ceiling is redundant.}\\
&= 3 + T(2^{m-1}) &&\\
&= 3 + 3 + T(2^{m-2}) &&\\
&= 3 + 3 + 3 + T(2^{m-3}) && \text{For each loop, add } +3 \text{ and subtract 1 from } m.\\
&= 3k + T(2^{m-k}) && \text{After } k \text{ substitutions.}\\
&= 3 + ... + 3 + T(2^{m-m}) && \text{After m substitutions, there are } m \text{ 3's.}\\
&= 3 + ... + 3 + T(1) && T(1) \text{ is the base case.}\\
&= 3m + 2. &&
\end{aligned}$$

  See proof by induction.

- $T(n) = 3\log_2 n + 2$.
  **Derivation:** If $T(2^m) = 3m+2$ for every number of students $n = 2^m, \log_2 n = m$. Substituting $m$ in, $T(n) = 3\log_2 n + 2$.

**Ceiling is redundant for integers.** You can drop the ceiling operation around an integer.

### 2.2.1 Prove $T(n) = 3m + 2,\ \forall n = 2^m$

**Claim:** Recurrence $T(1) = 2, T(n) = 3 + T(\lceil n/2 \rceil)$ has closed form $T(n) = 3m + 2$ for $\forall n \in \mathbb{N}, n = 2^m$.

*Proof by induction.* Let $H(m)$ be the statement $T(2^m) = 3m + 2$.
**Base:** LHS: $H(0) = T(2^0) = T(1) = 2$. RHS: $3(0) + 2 = 2$. LHS = RHS, so $H(0)$ is true.

**Inductive step:** Assume $H(m-1)$ is true to show $H(m)$ is true, that $T(2^m) = 3m + 2$. Since $H(m-1)$ is true, $T(2^m - 1) = 3(m-1) + 2$ by inductive hypothesis.

$$\begin{aligned}
T(2^m) &= 3 + T(\lceil (2^m/2) \rceil) \\
&= 3 + T(\lceil 2^m - 1 \rceil) \\
&= 3 + \boxed{T(2^m - 1)} && \text{Integer doesn't need ceiling.} \\
&= 3 + \boxed{3(m-1) + 2} && \text{By inductive hypothesis.} \\
&= 3 + 3m - 3 + 2 \\
&= 3m + 2
\end{aligned}$$

$\therefore H(m)$ is true. $\square$

## 2.3 Comparison

- **Simple counting:** $T(n) = 3n$ steps
- **Recursive counting:** $T(n) = 3\log_2 n + 2$ steps.

**Best:** Recursive counting.

Chapter 3

# Stable Matching

Stable Matching function

Assume there are an equal number $n$ of doctors and hospitals.
**Input:**

- $n$ doctors: $d_1 ... d_n$
- $n$ hospitals: $h_1 ... h_n$
- each doctor's ranking of hospitals (ex. $d_1 : h_2 > h_3 > h_1$)
- each hospital's ranking of doctors (ex. $h_1 : d_1 > d_3 > d_2$)

**Problem:** Create a stable assignment of doctors to hospitals, so no one switches jobs.
**Output:** The stable matching of doctors and hospitals.

**matching** $M$ A non-empty set of doctor-hospital pairs where no doctor/hospital appears twice.

**perfect matching** Every doctor/hospital appears once.
Terminology:

- "d is matched to h": $(d, h) \in M$
- "d is matched": $(d, h) \in M$ for some $h$

**unstable matching** Some doctor-hospital pair prefer one another to their mate in the matching.
Instabilities:

1. A matched doctor prefers an unmatched hospital. $d, h$ such that $d$ is matched to $h'$, $h$ is unmatched, but $d : h > h'$.
2. A matched hospital prefers an unmatched doctor. $d, h$ such that $h$ is matched to $d'$, $d$ is unmatched, but $h : d > d'$.
3. Two pairs want to swap partners. $d, h$ such that $d$ is matched to $h'$, $h$ is matched to $d'$, but $d : h > h'$ and $h : d > d'$.

Note that #1 and #2 cannot happen in a perfect match.

**stable matching** A perfect matching that has no instabilities.
There can be multiple stable matchings. A doctor/hospital may be with their 1st pick in one, and their 5th pick in another.
When there are different numbers of doctors and hospitals, the unmatched doctors/hospitals are still stable because no current pairs want to switch.
If all hospitals wanted the same doctor, assign that doctor to their 1st pick. This is never unstable, because that doctor would not want to leave.

## 3.1 Gale-Shapley Algorithm

Gale-Shapley for Stable Matching

```
Let M be empty
while some hospital h is unmatched do
    if h has offered a job to everyone then
        break                ▷ Return matchings. Does not apply if same number doctors/hospitals

        ▷ JOB OFFER ◁
    else let d be the highest-ranked doctor to which h has not yet offered a job
        h makes an offer to d
        if d is unmatched then
            d accepts, add (d,h) to M
        else if d is matched to h' & d: h' > h then
            d rejects, do nothing                          ▷ d prefers their current match
        else if d is matched to h' & d: h > h') then
            d accepts, remove (d,h') from M and add (d,h) to M   ▷ d dislikes their current match

Output M
```

**The party making offers (based in order of their preferences) get a stable matching optimal for them.** This is optimal for the hospitals (they get their best pick). (pareto-optimal). If doctors make offers to hospitals, results in a different stable matching that is optimal for doctors.
Assumes no ties in individual rankings.

Observations:

- **Hospitals make offers in descending order.** Hospitals make offers to doctors that they prefer less and less. Hospitals' happiness decreases during algorithm.
- **Doctors that get a job never become unemployed.** Once a doctor is matched, they never become unmatched with a hospital. They would only leave a hospital for another hospital. If a doctor is matched at some point, they are matched forever. Even if there were more doctors than hospitals, doctors never leave to become unemployed.
- **Doctors accept offers in ascending order.** Doctors receive offers that they prefer more and more. Doctors' happiness increase during algorithm.

### 3.1.1 Terminates after at most $n^2$ job offers

**Claim:** The GS algorithm terminates after at most $n^2$ iterations of the main loop ($n^2$ job offers). Loop stops when a hospital has made an offer to every doctor and was rejected.
Each hospital can offer each doctor at most 1 time = unique offers happen once. $n$ doctors $*$ $n$ hospitals = At most, there are $n^2$ unique offers.

### 3.1.2 Creates a perfect matching

**Claim:** The GS algorithm returns a perfect matching (all doctors/hospitals are matched).
*Assume equal number of doctors and hospitals, or perfect matching is impossible.*

*Proof by contradiction.* Suppose the claim is false: the Gale-Shapley algorithm does not return a perfect matching. Then there is some doctor $d$ which is not matched and also some hospital $h$ which is left out. $h$ is unmatched at the end of the algorithm, so $h$ has offered a job to everyone (included doctor $d$). So $h$ offered a job to $d$.

- Case 1: doctor $d$ accepts offer. By observation 2, doctors stay employed, so $d$ has to be matched at the end of the algorithm. Contradiction: $d$ cannot be unmatched, but is.
- Case 2: doctor $d$ rejects offer. Doctors only reject if they are already paired, and by observation 2, doctors stay employed. So doctor $d$ cannot be unmatched at the end. Contradiction: $d$ cannot be unmatched, but is.

□

### 3.1.3 Creates a stable matching

**Claim:** The GS algorithm returns a stable matching.

*Proof by contradiction.* Suppose the claim is false: the Gale-Shapley algorithm does not return a stable matching. Then there is an instability in the matching $M$: $(d, h'), (d', h)$, such that $h : d > d'$ and $d : h > h'$. Since $h$ prefers $d$, we know $h$ made an offer to $d$ before $d'$.

- Case 1: $d$ rejected the offer. Then we know $d$ prefers their current hospital $h*$ to hospital $h$, that $d : h* > h$. Putting this together, $d : h* > h > h'$. By observation 3, doctors get happier as algorithm progresses. If doctor $d$ ended up with $h'$, then $d$ must prefer $h'$ over $h*$, so $d : h' > h* > h$. Contradiction: $d : h* > h > h' \rightarrow\!\times\!\leftarrow d : h' > h* > h$.
- Case 2: $d$ accepted the offer. $(d, h) \in M$ at some point. By observation 3, doctors get happier. Since $d$ ended up with $h'$, then $d : h' > h$. This contradicts $d : h > h'$. Contradiction: $d : h' > h \rightarrow\!\times\!\leftarrow d : h > h'$.

□

### 3.1.4 Total number of operations

- Doctors' preferences stored as ranked list of hospitals: In the worst case, uses $n^3$ operations. Loop runs $\leq n^2$ times $= n^2$ job offers. Each offer uses $\leq n$ operations: $h$ has to be found in the doctor's list of preferences.
- Doctors' preferences stored as list of rankings indexed by hospital: In the worst case, uses $n^2$ operations. Each offer uses 1 operation since the position in the list is known.

# Chapter 4

# Sorting a List

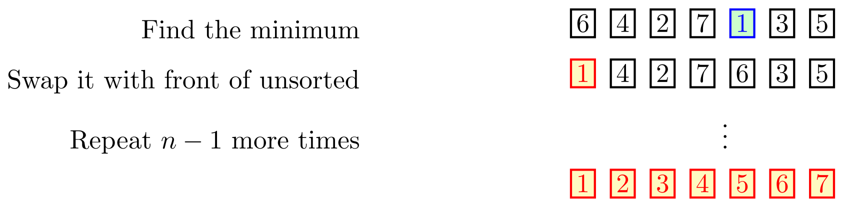
**Sorting function**

**Input:** A list of $n$ comparable elements (numbers, words, ...).
**Problem:** Sort the list so that it is ordered.
**Output:** The list of elements in ascending (or descending order).

## 4.1 Selection Sort

**Selection sort**

```
1: for $i = 1, \ldots, n$ do
2:     Look through unsorted portion of list, A[i+1, n]. Find the minimum.    ▷ O(n)
3:     Swap the minimum with element at $i$                                   ▷ O(1)
```



**Observations:**

- The runtime is $O(n^2)$. Scanning list for minimum is $O(n)$. Swapping doesn't affect the algorithm asymptotically, since it is always 1 as $n$ increases (doesn't depend on $n$). Repeats on list of size $n-1, n-2, n-3, \ldots$. Total number of operations $= n + n - 1 + n - 2 + \ldots + 1 = \sum_{i=1}^{n} i = O(n^2)$. If you count swaps, there are $n$ swaps, and each swap takes a constant number of operations. $n^2$ for scans $+\ cn$ for swaps, can drop lower order terms. The dominant behavior is the scans.

## 4.2 Mergesort

**Summary:**

- Sorts a list of $n$ objects in $\Theta(n \log n)$ time.
- Can sort any that allows comparisons.
- No comparison based algorithm can be significantly faster.

1. Split list in half. If there are odd elements, add a ceiling and floor to algorithm.
2. Keep splitting and recursively sort.
3. Merge pieces back together. Compare the first elements of each list.

MergeSort

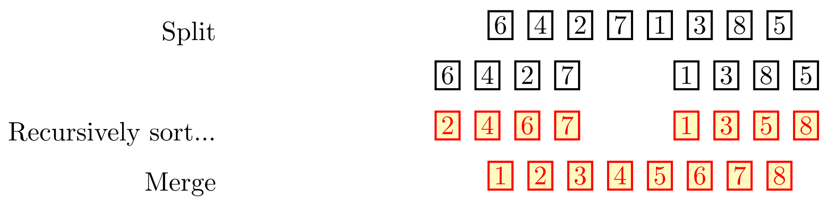
```
function MergeSort(A)
    if len(A) = 1 then                                  ▷ Base case
        return A

    Let $m \leftarrow \lceil len(A)/2 \rceil$                          ▷ Split
    $L \leftarrow A[1:m], R \leftarrow A[m+1:n]$

    Let $L \leftarrow MergeSort(L)$                       ▷ Recurse
    Let $R \leftarrow MergeSort(R)$

    $A \leftarrow Merge(L, R)$                 ▷ Merge L and R into a single list A

    Return $A$
```



**Explanation:** If $L$ and $R$ are sorted lists of length $\frac{n}{2}$, we can merge them into a sorted list $A$ of length $n$ in time $\Theta(n)$, since $n-1$ comparisons to fill a list of size $n$ (last item left needs no comparison). **Merging 2 sorted lists is faster than sorting from scratch.**
(Best case: If the largest element in one half is smaller than the smallest element in the other half, you might only need $\frac{n}{2}$ comparisons, a constant * $n$, still linear, with additional copying for the 2nd half of the list.)
**Observations:**

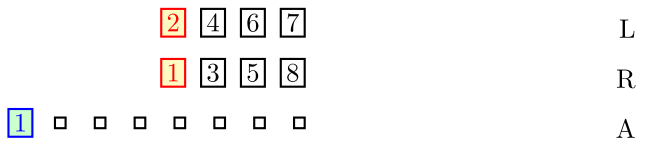
- Runtime on list of size $n$: $T(1) = O(1)$. $T(n) = 2T(\frac{n}{2}) + cn$. Since `T` amd `R` are size $\frac{n}{2}$, `MergeSort(T)` and `MergeSort(T)` each take $T(\frac{n}{2})$. `Merge(L,R)` takes $O(n) = cn$. `Return A` takes $O(1) = c$. Assigning $m, L, R$ take $O(n)$ to copy $n$ elements into $L$ or $R$. Non-recursive parts of algorithm: $c + cn + cn + c = O(n)$. Therefore, $T(n) = 2T(\frac{n}{2}) + O(n) = 2T(\frac{n}{2}) + cn$ (change asymptotic analysis notation to standard form to perform arithmetic on it).

Merge

```
function MERGE(L,R)
    Let $n \leftarrow len(L) + len(R)$
    Let $A$ be an array of length $n$
    $j \leftarrow 1, k \leftarrow 1,$
    ▷ $j$ is the index variable for L, $k$ is the index variable for R ◁

    for $i = 1, \ldots, n$ do
        if $j > len(L)$ then                                  ▷ L is empty. Put next R item
            $A[i] \leftarrow R[k], k \leftarrow k + 1$
        else if $k > len(R)$ then                             ▷ R is empty. Put next L item
            $A[i] \leftarrow L[j], j \leftarrow j + 1$
        else if $L[j] \leq R[k]$ then      ▷ L is smallest. Flip operator to sort descending
            $A[i] \leftarrow L[j], j \leftarrow j + 1$
        else                                                  ▷ R is smallest
            $A[i] \leftarrow R[k], k \leftarrow k + 1$

    Return $A$
```



**Observations:**

- Runtime is $\Theta(n)$. The *For* loop repeats $n$ times. In the worst case (last branch), there are 3 comparisons + 2 assignments = 5 operations. Total worst case: $5n + 4 = O(n)$. In the best case (first branch), there are 3 operations, $3n + 4 = \Omega(n)$.

### 4.2.1 Prove Correctness of Mergesort

**Claim:** The algorithm `Mergesort` is correct.

*Proof by Strong Induction.* Let $H(n)$ be the statement that `Mergesort` correctly sorts all lists of length $n$.
**Base:** $H(1)$ directly follows from the algorithm. A list of size 1 is already sorted.
**Inductive step (assume we have already argued correctness of Merge):** Assume $H(1) \wedge H(2) \wedge \ldots \wedge H(k-1)$. Show $H(k)$.
L and R have length $\leq \lceil \frac{k}{2} \rceil$. $\lceil \frac{k}{2} \rceil < k$ when $k \geq 2$, so $\lceil \frac{k}{2} \rceil$ is in the chain. $\therefore$ By IH, L and R are correctly sorted.
Since `Merge` properly merges 2 sorted lists into 1 sorted list, we know A is sorted. □

### 4.2.2 Solving Mergesort Recurrence

**Recurrence:** $T(n) = 2 * T(n/2) + Cn$. $T(1) = C'$. Each list is split into 2 subproblems of size $n/2$.

| Level | Piece size | Tree | # of pieces | Work @ level |
|---|---|---|---|---|
| 0 | $n$ | $\boxed{n}$ | $1 = 2^0$ | $cn$ |
| 1 | $\frac{n}{2}$ | $\boxed{\frac{n}{2}}\ \boxed{\frac{n}{2}}$ | $2 = 2^1$ | $2\frac{cn}{2} = cn$ |
| 2 | $\frac{n}{4}$ | $\boxed{\frac{n}{4}}\ \boxed{\frac{n}{4}}\ \boxed{\frac{n}{4}}\ \boxed{\frac{n}{4}}$ | $4 = 2^2$ | $4\frac{cn}{4} = cn$ |
| i | $\frac{n}{2^i}$ | | $2^i$ | $2^i\frac{cn}{2^i} = cn$ |
| $\log_2 n$ (last level) | | | | |

Finding the last level: Set generic piece size formula = base case. Solve for $i$.

$$\begin{aligned} \frac{n}{2^i} &= 1 \\ n &= 2^i \\ \log_2 n &= i && \text{Last level.} \end{aligned}$$

**Total work:** $\sum_{i=0}^{\log_2 n} cn = (\log_2 n + 1)cn = O(n \log n)$. There are $\log_2 +1$ terms (including $i = 0$) and each term is $cn$.

# Chapter 5

# $n$-digit Multiplication

## 5.1 $n + n$ Addition

Given $n$-digit numbers $x, y$, output $x + y$.

| | | | | |
|---|---|---|---|---|
| | 1 | 2 | 3 | 4 |
| + | 1 | 1 | 2 | 2 |
| = | 2 | 3 | 4 | 6 |

**Observations:**

- Running time: $O(n)$. There are $n$ additions between 2 digits and $\leq n - 1$ carries.

## 5.2 $n * n$ Multiplication

Given $n$-digit numbers $x, y$, output $x * y$.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | | | 1 | 2 | 3 | 4 |
| | | | * | 1 | 1 | 2 | 2 |
| | | | | 2 | 4 | 6 | 8 |
| + | | | 2 | 4 | 6 | 8 | 0 |
| + | | 1 | 2 | 3 | 4 | 0 | 0 |
| + | 1 | 2 | 3 | 4 | 0 | 0 | 0 |
| | 1 | 3 | 8 | 4 | 5 | 4 | 8 |

- Running time: $O(n^2)$. There are $n^2$ multiplications between 2 digits, resulting in $n$ numbers with $O(n)$-digits that must be added together ($n - 1$ additions). We know addition takes constant time, so addition takes $cn * (n - 1) = O(n^2)$. $n^2 + cn^2 = O(n^2)$.

## 5.3 Divide and Conquer Multiplication

For factors of the same, even length. Partition each factor in half. If a factor has an odd/different number of digits, prepend the front with 0's.

### 5.3.1 Derivation

| | first n/2 digits | second n/2 digits |
|---|---|---|
| | **a** | **b** |
| * | **c** | **d** |

$$\begin{aligned}
\mathbf{x} = \{\mathbf{ab}\} &= a10^{n/2} + b \\
\mathbf{y} = \{\mathbf{cd}\} &= c * 10^{n/2} + d \\
\mathbf{x*y} &= (a * 10^{n/2} + b)(c * 10^{n/2} + d) && \text{FOIL.} \\
&= ac * 10^{n/2} + (ad + bc)10^{n/2} + bd
\end{aligned}$$

Instead of $n * n$ division, we are doing four $\frac{n}{2} * \frac{n}{2}$ multiplications, three $n$-digit additions, and some shifts. $*10$ is easy because its just shifting by 0.
**Running time:** $T(n) = 4T(\frac{n}{2}) + \Theta(n)$. $T(1) = 1$. The multiplications are recursive, the adds and shifts each $O(n)$.

### 5.3.2 Prove $T(n) \geq n^2$

**Claim:** $T(n) \geq n^2$. Divide and Conquer multiplication is not any better than the $n * n$ division algorithm.

*Proof by strong induction.* **Base:** $H(1)$. LHS $= T(1) = 1$. RHS $= 1^2 = 1$. LHS $=$ RHS. The base case holds true.
**Inductive step:** Need strong induction for $k/2$ term. Assume the claim is true for $H(1) \wedge H(2) \wedge ... \wedge H(k-1) \implies H(k)$.
By IH, $H(\frac{k}{2})$ is true, meaning $T(\frac{k}{2}) \geq (\frac{k}{2})^2 = \frac{k^2}{4}$. Show $T(k) \geq k^2$.

$$\begin{aligned}
T(k) &= 4T(\frac{k}{2}) + Ck \\
&\geq 4T(\frac{k^2}{4}) + Ck && \text{By IH.} \\
&= k^2 + Ck \\
&\geq k^2.
\end{aligned}$$

□

## 5.4 Karatsuba's Algorithm

**Summary:**

- A multiplication algorithm that's faster than $n^2$. Like Divide and Conquer Multiplication, but uses 1 less multiplication. **Key fact:** Adding is faster than multiplying.
- Multiplies $n$-digit numbers in $O(n^{1.59})$ time.
- Fast Fourier Transform is even faster: $\approx O(n \log n)$ time.
- For rest of class, we assume our inputs have $O(1)$ digits. Multiplication is treated as a $O(1)$ operation.

**Explanation:**
Divide and Conquer Multiplication: $x * y = ac * 10^{n/2} + (bc + ad)10^{n/2} + bd$
Key identity: $(b - a)(c - d) = bc - bd - ac + ad$
$(b - a)(c - d) + bd + ac = bc + ad$.
Only uses three $n/2$-digit multiplications (plus adds and shifts):

1. $ac$
2. $bd$
3. $(b-a)(c-d)$

**Karatsuba's Algorithm**

```
function KARATSUBA(x,y,n)
    if $n = 1$ then                                    ▷ Base case
        return $x * y$

    Let $m \leftarrow \lceil n/2 \rceil$                ▷ Split
    Write $x = 10^m a + b, y = 10^m c + d$              ▷ Assign a,b,c,d

    Let $e \leftarrow Karatsuba(a, c, m)$               ▷ Recurse
    Let $f \leftarrow Karatsuba(b, d, m)$
    Let $g \leftarrow Karatsuba(b-a, c-d, m)$

    Return $10^{2m} e + 10^m (e + f + g) + f$           ▷ Merge
```

Store digits in an array or use strings... (The exact container is language-specific).
**Running Time:** $T(n) = 3T(\frac{n}{2}) + cn$. $T(1) = c$. Assigning $x$ and $y$ by copying each of the n digits into $a, b, c$, or $d$ is )$O(n)$. 3 multiplications = 3 recursive calls. 2 shifts = adding on n 0's = $O(n)$. 4 additions = $O(n)$.

### 5.4.1 Prove: Correctness of Karatsuba

**Claim:** The algorithm `Karatsuba` is correct.

*Proof.* Let $H(n)$ be the statement `Karat(x,y,n)` is correct.
**Base:** O(1). **Inductive step:** $H(1) \wedge H(2) \wedge ... \wedge H(k-1) \implies H(k)$.
$\frac{k}{2} < k$, so $H(\frac{k}{2})$ appears in the chain. $H(\frac{k}{2})$ is true by IH. $a, b, c, d, b-a, c-d$ all have $< k$ digits. Therefore, $e = ac$ and $f = bd$ and $g = (b-a)(c-d)$ are all correct by IH (the multiplication is correct). Then the algorithm $karat(x, y, n) = 10^k e + 10^{k/2}(e + f + g) + f = 10^k ac + 10^{k/2}(bc + ad) + bd$, resulting in the divide and conquer multiplication formula. □

### 5.4.2 Solving Karatsuba Reccurence

**Recurrence:** $T(n) = 3T(\frac{n}{2}) + cn$.
**Recursion tree:**

| Level | Piece size | Tree | # of pieces | Work @ level |
|---|---|---|---|---|
| 0 | $n$ | $\boxed{n}$ | $1 = 3^0$ | $cn$ |
| 1 | $\frac{n}{2}$ | $\boxed{\frac{n}{2}}\boxed{\frac{n}{2}}\boxed{\frac{n}{2}}$ | $3 = 3^1$ | $3\frac{cn}{2} = \frac{3}{2}cn$ |
| 2 | $\frac{n}{4}$ | $\boxed{\frac{n}{4}}\boxed{\frac{n}{4}}\boxed{\frac{n}{4}}\boxed{\frac{n}{4}}\boxed{\frac{n}{4}}\boxed{\frac{n}{4}}\boxed{\frac{n}{4}}\boxed{\frac{n}{4}}\boxed{\frac{n}{4}}$ | $9 = 3^2$ | $9\frac{cn}{4}$ |
| i | $\frac{n}{2^i}$ | | $3^i$ | $3^i\frac{cn}{2^i} = cn(\frac{3}{2})^i$ |
| $\log_2 n$ (last level) | | | | |

Finding the last level: Set generic piece size formula = base case size of 1. Solve for $i$.

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

$$n = 2^i$$

$$\log_2 n = i \qquad \text{Last level.}$$

**Total work:** $\sum_{i=0}^{\log_2 n} cn(\frac{3}{2})^i$.

$$\sum_{i=0}^{\log_2 n} cn(\frac{3}{2})^i \qquad \text{Since } cn \text{ is independent of } i\text{, pull it out of summation.}$$

$$cn\sum_{i=0}^{\log_2 n} (\frac{3}{2})^i \qquad \text{This is a geometric series. } r = 3/2 > 1.$$

Asymptotic behavior: $cn * c'(\frac{3}{2})^{\log_2 n}$ By property of a geometric series, $r > 1$.

$$= cn * c'(\frac{3^{\log_2 n}}{2^{\log_2 n}})$$

$$= cn * c'(\frac{3^{\log_2 n}}{n})$$

$$= c * c' 3^{\log_2 n}$$

$$= O(n^{\log_2 3} = n^{1.59})$$

# Chapter 6

# Array Searching

Find the index of a target value $t$ in this list $A$ of size $n$.

## 6.1 Binary Search

**Input:** Sorted list.
**Summary:**

- Searches a sorted array in $O(\log n)$ time.
- **Key fact:** eliminate half the list each time.

**Explanation:** If $t$ were in the list, which half would it be in? Let $l$ be the first index and $r = n$ be the last index. Compare $t$ with the middle value at $m = \lfloor (r - l)/2 \rfloor$). If $t \leq m$, look in left half. If $t > m$, look in right half. Keep recursing.
When comparing $t$ vs $A[m]$, there are 3 cases:

1. $t < A[m]$. Search in $[l, ..., m - 1]$.
2. $t = A[m]$. Return $m$.
3. $t > A[m]$. Search in $[m + 1, ..., r]$.

Binary Search

```
function SEARCH(A,t)
    ▷ A[1:n] sorted in ascending order ◁
    Return $BS(A, 1, n, t)$
function BS(A,l,r,t)
    if $l > r$ then
        Return FALSE                    ▷ Left goal post passed over right goal post

    Let $m \leftarrow l + \lfloor \frac{r-l}{2} \rfloor$

    if $A[m] = t$ then                  ▷ Found target
        Return $m$
    else if $A[m] > t$ then             ▷ Look in left half
        Return $BS(A, l, m - 1, t)$
    else                                ▷ Look in right half
        Return $BS(A, m + 1, r, t)$
```

**Observations:**

- **Recurrence:** $T(n) = T(n/2) + c$ in the worst case. $T(1) = O(1)$. It's possible the target value is in the middle $= \Omega(1)$. In the *If/elseif/else* statement, only recursive call happens. Recurses on half the list, $n/2$. All other operations are constant time $= O(1)$.
- **Running time:** $\Theta(\log n)$. Solving the recurrence using the Master Theorem: $a = 1, b2, d = 0. \frac{a}{b^d} = \frac{1}{2^0} = 1$. So $T(n) = \Theta(n^0 \log n) = \Theta(\log n)$ is the worst running time.

# Chapter 7

# Selection

Given an array of $n$ numbers, $A[1,...,n]$, how quickly can I find the:

- Smallest number? $T(n) = O(n)$.
- Second smallest? $T(n) = O(n)$. Scan $n$ for first smallest, then scan $n-1$ for second.
- $k$-th smallest? $kn = O(kn)$. $k$ is a variable, and NOT a constant. If $k$ is specified to be a constant, like $k = 5$, you can drop it, but if $k = n/2$, you cannot.
- median? $k = n/2$. $n/2 * n$, $1/2$ is a constant, drop it, $O(n^2)$. In this case, it would be more efficient to sort the list and get the median.

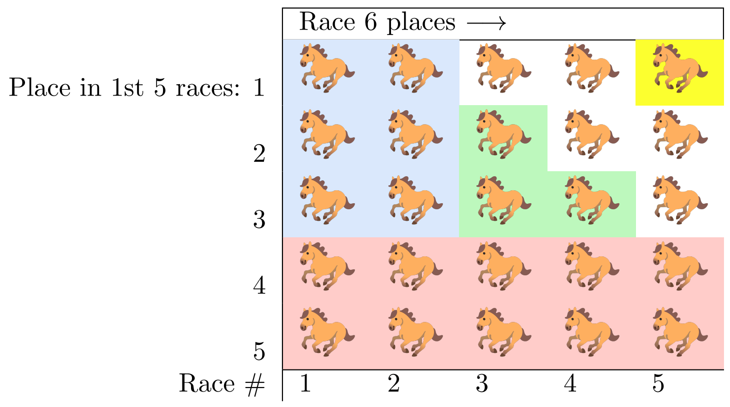
**Observations:**

- Can select the $k$-th smallest in $O(n \log n)$ time. Mergesort the list, then look up $A[k]$.
- The Median algorithm can select the $k$-th smallest in $O(n)$ time.

### 7.0.1 Warmup: Finding 3 fastest horses

**Problem:** You have 25 horses and want to find the 3 fastest in only 7 races. You have a racetrack were you can race 5 at a time. You don't have a stopwatch. Each horse has a consistent finish time.
**Solution:** Discard horses a candidates for being the top 3.

1. Race all 25 horses in 5 races.
2. (In red) Discard the 2 losers in each race = 15 horses left.
3. Race the winners together in a sixth race. The winner is the #1 horse (in yellow).
4. (In blue) Discard the 2 losers of race six, who were beaten by 3 horses. Discard the horses slower than those 2 losers.
5. Discard the horses in green, who were beaten by at least 3 horses.
6. Race the remaining uncolored horses to determine #2 and #3 = 7 races total.



## 7.1 Median Algorithm: Version 1

**partition around the pivot** Compare every element against the pivot. If smaller than pivot, put to left. If larger than pivot, put to right.

After partitioning around the pivot, we know the accurate location of the pivot in the list.

| | |
|---|---|
| Before partition around the pivot | $\boxed{4}\ \boxed{2}\ \boxed{1}\ \boxed{5}\ \boxed{3}$ |
| After partition around the pivot | $\boxed{2}\ \boxed{3}\ \boxed{1}\ \boxed{4}\ \boxed{5}$ |

Median Algorithm: Take 1

```
function SELECT(A[1:n], k)
    if $n = 1$ then
        Return $A[1]$

    Choose a pivot $p = A[1]$                          ▷ Random pivot, like 1st element
    Partition around the pivot, let $r = indexOf(A, p)$

    if $k = r$ then
        Return $A[r]$                                  ▷ $k^th$ smallest element
    else if $k < r$ then
        Return $Select(A[1 : r - 1], k)$               ▷ Must be in left
    else if $k > r$ then
        Return $Select(A[r + 1 : n], k - r)$           ▷ Must be in right
```

**Observations:**

- This is bad if the input is sorted, and we want the last elements. We only eliminate 1 element at a time (the pivot). Worst case: $O(n^2)$ when everything is to 1 side of the pivot:

  $\boxed{1}\ \boxed{2}\ \boxed{3}\ \boxed{4}\ \boxed{5}\ \boxed{6}\ \boxed{7}$

  $\boxed{2}\ \boxed{3}\ \boxed{4}\ \boxed{5}\ \boxed{6}\ \boxed{7}$

  $\boxed{3}\ \boxed{4}\ \boxed{5}\ \boxed{6}\ \boxed{7}$

$\boxed{4}$ $\boxed{5}$ $\boxed{6}$ $\boxed{7}$
Takes $n/2$ pivots, and partitioning around the pivot each time is $O(n)$.

- Similar to quick sort, but doesn't sort the entire list.

## 7.2 Median Algorithm: Version 2

**Problem:** Finding a good pivot element. **Solution:** The best pivot would be the median, then we discard half the list each time. Finding the median is hard, but it is good enough to use the "median of medians" instead, a value that is "close to the median".

### 7.2.1 Median of Medians (MOM)

Median of Medians

```
function MOM(A[1:n])
    ▷ Split list into groups of 5. $m$ is number of groups and length of $M$ ◁
    Let $m \leftarrow \lceil n/5 \rceil$
    for $i = 1, \dots, m$ do
        $M[i] \leftarrow medianA[5i-4], \dots, A[5i]$
        ▷ Find median of each group. $M$ is list of medians. ◁
    $p \leftarrow Select(M[1:m], \lceil m/2 \rceil)$    ▷ Call Select(list of medians, median index) to find MOM
    Return p    ▷ The median of medians
```

`MOM` **running time:** Finding the median in list of length $5 = O(1)$ since the size of list is constant and sorting it takes a constant number of operations. Find the median $n/5$ times $= O(n)$. To find $M$, $n/5$ groups $*c$ work to find the median in a group of $5 = O(n)$. Then `MOM` running time is $cn+$ time to run `Select` on a list of size $m = n/5$. If $n < 5$, there's just one group, and can find median of that group easily.

#### Prove: The Median of Medians is a good pivot

**Claim:** For every $A$ there are at least $\frac{3n}{10}$ items (30% of the list) that are smaller than $MOM(A)$.

*Proof.* There are $\frac{n}{5}$ groups total. In half of those $\frac{n}{5}$ groups, $MoM \geq 3$ elements, so in $\frac{n}{q0}$ groups, MoM is $\geq 3$ elements.
$\therefore$ MoM is $\geq \frac{3n}{10}$ elements. MoM is also $\leq \frac{3n}{10}$ elements. So MoM is in the middle 40% of all elements. □

By using the MoM, you are always able to throw out at least 30% of the list with each recursive step, even in the worst case. **Note:** 5 is the smallest group size that gives a running time of $O(n)$. Groups of size 3 scale worse.

### 7.2.2 Finding Recurrence

**Median Algorithm: Version 2 with Median of Medians**

```
▷ To find k-th smallest element ◁
function MOMSelect($A[1:n]$,$k$)
    if $n \le 25$ then
        ▷ Small n, brute force it ◁
        Sort $A$ and return $A[k]$

    Let $p = MOM(A)$                                   ▷ MOM calls MOMSelect (recursive)
    Partition around the pivot, let $r = IndexOf(A, p)$

    if $k = r$ then
        Return $A[r]$
    else if $k < r$ then
        return MOMSelect(A[1:r-1], k)
    else if $k > r$ then
        return MOMSelect(A[r+1:n], k-r)
```

If $n$ is small, brute force finding $k$ by sorting the list $= O(1)$ because the group has constant size ($n \le 25$). You could check if $n \le 1000000$, would still be constant time because it performs the same as $n$ gets very large. If you set $n =$ something in terms of $n$, cannot guarantee linear time.

**If you set $n$ to a constant then the time complexity is linear.** If you set $n =$ something in terms of $n$, cannot guarantee linear time.

**Finding recurrence:** The total time complexity of MOM is $\Theta(n) + T(n/5)$. We found above the time complexity of MOM is $\Theta(n)$ for operations excluding the recursive call. Let $T(n)$ be the time to run `MOMSelect` on size $n$. There are $n/5$ groups, so $T(n/5)$ is the time to find MoM.
Partitioning around the pivot $= O(n)$. In the worst case, we eliminate at least $3n/10$ of the list, so at most we recurse on $T(7n/10)$. Then in total, $T(n) = cn + T(\frac{n}{5}) + T(\frac{7n}{10})$. $T(1) = O(1)$.

### 7.2.3 Solving Recurrence:

Doesn't fit the Master Theorem. Must use Recursion Tree. **Recurrence:** $T(n) = cn + T(\frac{n}{5}) + T(\frac{7n}{10})$. $T(1) = O(1)$. **Recursion Tree:**

| Level | Largest piece size | Tree | # of pieces | Work @ level |
|---|---|---|---|---|
| 0 | $n$ | $\boxed{n}$ | | $cn$ |
| 1 | $\frac{7n}{10}$ | $\boxed{\frac{7n}{10}}\boxed{\frac{2n}{10}}$ | | $cn(\frac{7}{10} + \frac{2}{10})$ |
| 2 | $\frac{49n}{100}$ | $\boxed{\frac{49n}{100}}\boxed{\frac{14n}{100}}\boxed{\frac{14n}{100}}\boxed{\frac{4n}{100}}$ | | $cn(\frac{81}{100})$ |
| i | $\left(\frac{7}{10}\right)^i n$ | | | $cn(\frac{9}{10})^i$ |
| ... | | | | |
| $\log_b n$ (last level) | | | | |

Work at each level comes from non-recursive terms: $cn$.
**Total work across all levels:** $\sum_{i=0} cn(\frac{9}{10})^i = cn\sum_{i=0}(\frac{9}{10})^i = cnc' = O(n)$ running time.
We didn't need to find the last level or the piece size because the series is geometric, $r = \frac{9}{10} < 1$, and converges to a constant: $c'$. If you wanted to write the accurate summation, use the largest piece size for each level and set the general formula equal to the base case. Solving $\left(\frac{7}{10}\right)^i n = 1$ results in $i = \log_{10/7} n$ as

the last level.

### 7.2.4 Finding MOM with n/3 groups of size 3 is worse

**Items eliminated:** ATTEMPT: There are $n/3$ groups. In half the groups, MoM is $\geq$ 2 elements $= 2n/6 = n/3$ items.
**New running time:** ATTEMPT: $T(n) = cn + T(n/3)$ for MOM $+T(2n/3)$ items left to check.
Work at each level: $cn, cn(1/3 + 2/3) = cn, cn(1/9 + 2/9 + 2/9 + 4/9) = cn$, is always $cn$.
Largest piece: $n, 2n/3, 4n/9, (2/3)^i * n$.
Find last level: $(2/3)^i = 1, i = log_{3/2}(n)$.
Then the running time is $O(cn \log_{3/2} n)$, which is worse than constant time!

## 7.3 Summary

- Can find the $k$-th largest/smallest element in $O(n)$ time.
- Selection is easier than sorting (Mergesort is $O(n \log n)$).
- **Key fact:** A Median of Medians of 5 is a good pivot point. Building the list of medians and partitioning around pivot are each $O(n)$. So choosing larger groups won't be better than $O(n)$.
- Similar pivot technique can be used to sort in $O(n \log n)$: `Quicksort`.
- A random pivot is also a good pivot in expectation. (Random pivot can give a O(n) algorithm, but not guaranteed (if pivot is the first item)).

# Chapter 8

# Weighted Interval Scheduling

Weighted Interval Scheduling function

**Input:** $n$ intervals $(s_i, f_i)$ each with value $v_i$.
Assume intervals are sorted: $f_1 < f_2 < \cdots < f_n$.
**Problem:** Optimally schedule a resource to $n$ people who want to use it at a given start and end time, depending on how much they will pay you.
**Output:** A compatible schedule $S$ **maximizing** the total value of all intervals.

**Terminology:**

- **schedule:** a subset of intervals $S \subseteq \{1, ..., n\}$.
- A schedule is **compatible** if no $i, j \in S$ overlap.
- The **total value** of $S$ is $\sum_{i \in S} v_i$, the sum of all values in the schedule. This is maximized.

## 8.1 Recurrence

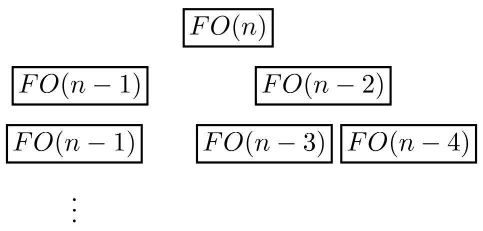
Let $O$ be the *optimal* schedule. Look at the end (first or last interval), not the middle. If we start with a middle interval, the subproblems are separated.
**Subproblems:** Let $O_i$ be the **optimal schedule** using only the first $i$ intervals: $\{1, ..., i\}$.

1. **Case 1:** Final interval is not in $O_i$ ($i \notin O_i$). Then $O_i = O_{i-1}$, the optimal solution for the rest of the intervals: $\{1, ..., i-1\}$.
2. **Case 2:** Final interval is in $O_i$ ($i \in O_i$). Assume intervals are sorted by finish time: $f_1 < f_2 < \cdots < f_n$. Let $p(i)$ be the last interval that finishes before $i$ begins, such that $f_j < s_i$ for the largest $j$. Then $O_i$ must contain $i$ and the optimal solution for intervals that don't overlap with $i$: $\{1, ..., p(i)\}$. So $O_i = \{i\} + O_{p(i)}$.

So the subproblems are: $O_{i-1}, O_{p(i)}$. To choose the better one, we need to find their value.
Let $OPT(i)$ be $value(O_i)$, the ***value* of the optimal schedule** using only the first $i$ intervals: $\{1, ..., i\}$.
**Recurrence:** $OPT(i) = max\{OPT(i-1), v_i + OPT(p(i))\}$, max(case1, case2). **Bases:** $OPT(0) = 0, OPT(1) = v_1$

**Find Optimal Schedule without DP**

```
▷ All inputs are global vars ◁
function FindOPT(n)
    if $n = 0$ then
        Return 0
    else if $n = 1$ then
        Return $v_1$
    else
        Return $max\{FindOPT(n-1), v_n + FindOPT(p(n))\}$
```



**Running time in worst case:** exponential when intervals overlap like a staircase. There are $n$ calls in the left branch and the right branch forks.

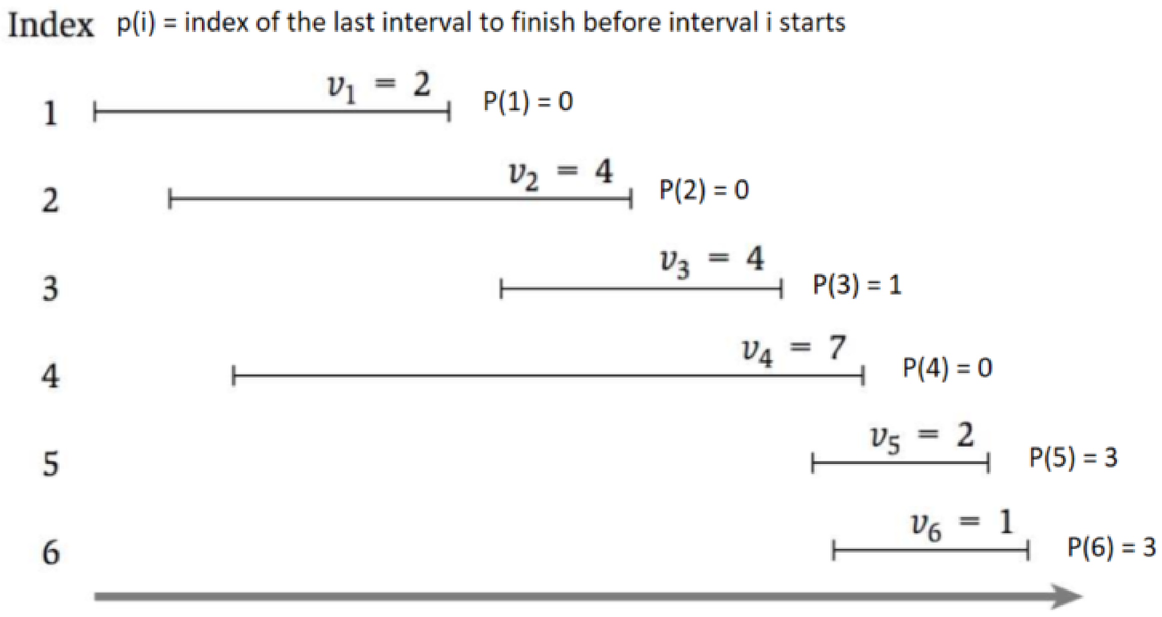
## 8.2 Interval Scheduling: Top Down

**Find Optimal Schedule with Top Down DP**

```
▷ All inputs are global vars ◁
$M \leftarrow$ empty array, $M[0] \leftarrow 0, M[1] \leftarrow v_1$
function FindOPT(n)
    if $M[n]$ is not empty then                          ▷ Already computed
        Return $M[n]$
    else                                                 ▷ Solve recurrence
        $M[n] \leftarrow max\{FindOPT(n-1), v_n + FindOPT(p(n))\}$
        Return $M[n]$
```

We want any optimal schedule, so break ties between cases arbitrarily.
**Running time:** $n-1$ elements to fill. Each fill needs 2 recursive calls. We compute each element 1 time, and we only do recursive calls when computing a new element. Total: $2(n-1) = O(n)$.

**Example:**



| $M[0]$ | $M[1]$ | $M[2]$ | $M[3]$ | $M[4]$ | $M[5]$ | $M[6]$ |
|---|---|---|---|---|---|---|
| 0 | 2 | 4 | 6 | 7 | 8 | 8 |

$M[i]$ is the optimal value using the 1st $i$ intervals.

Top down: start from high values to low values of $i$.
Chain of recursive calls:
$FO(6) : FO(5) vs FO(3) + v_6$
$FO(5) : FO(4) vs. FO(3) + v_5$
$FO(4) : FO(3) vs. FO(0) + v_4$
$FO(3) : FO(2) vs FO(1) + v_3$
$FO(2) : FO(1) vs. FO(0) + v_2$
Values are in table.
$FO(2) = 2 vs. 0 + 4 = 4.$
$FO(3) = 4 vs. 2 + 4 = 6.$
$FO(4) = 6 vs. 0 + 7 = 7.$
$FO(5) = 7 vs. 6 + 2 = 8.$
$FO(6) = 8 vs. 6 + 1 = 8.$

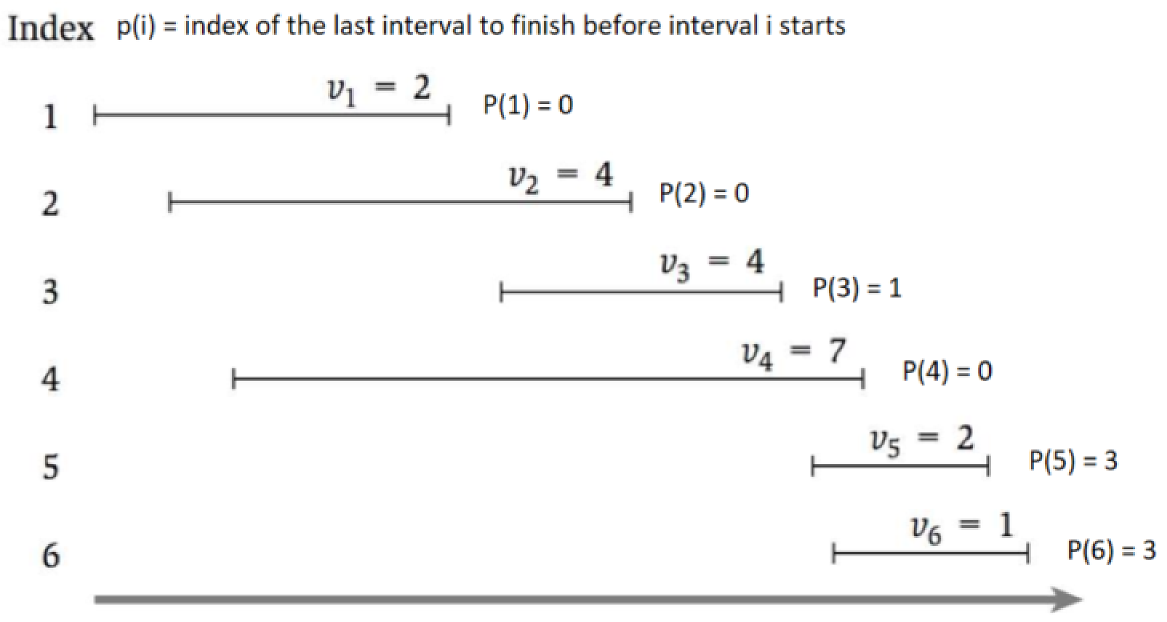
## 8.3 Interval Scheduling: Bottom Up

**Find Optimal Schedule with Bottom Up DP**

```
1: ▷ All inputs are global vars ◁
2: function FINDOPT(n)
3:     $M[0] \leftarrow 0, M[1] \leftarrow v_1$
4:     for $i = 2, ..., n$ do                              ▷ Iterate
5:         $M[i] \leftarrow max\{M[i-1], v_i + M[p(i)]\}$
6:     Return $M[n]$                                       ▷ $OPT(n)$
```

**Running time:** $n - 1$ iterations in *For* loop, each iteration does constant time operations. Total: $O(n)$.

**Example:**



| $M[0]$ | $M[1]$ | $M[2]$ | $M[3]$ | $M[4]$ | $M[5]$ | $M[6]$ |
|---|---|---|---|---|---|---|
| 0 | 2 | 4 | 6 | 7 | 8 | 8 |

$M[i]$ is the optimal value using the 1st $i$ intervals.

Bottom up: start from low values to high values of $i$.
$FO(0) = 0$.
$FO(1) = 0vs.0 + 1 = 1$.
$FO(2) = M[1]vs.M[0] + v_2.2vs.0 + 4 = 4$.
$FO(3) = M[2]vs.M[1] + v_3.4vs.1 + 4 = 5$.
$FO(4) = M[3]vs.M[0] + v_4.5vs.0 + 7 = 7$.
$FO(5) = M[4]vs.M[3] + v_5.7vs.4 + 4 = 8$.
$FO(6) = M[5]vs.M[3] + v_6.8vs.5 + 1 = 8$.

## 8.4 Reconstructing the Optimal Solution from DP table

After DP table is filled in...

Reconstruct solution from filled DP table

```
▷ All inputs are global vars ◁

▷ Find optimal schedule on first n intervals ◁
function FINDSCHED(M,n)
    if $n = 0$ then
        Return $\emptyset$
    else if $n = 1$ then
        Return $\{1\}$
    else if $v_n + M[p(n)] > M[n-1]$ then                ▷ Case 2
        Return $\{n\} + FindSched(M, p(n))$
    else                                                 ▷ Case 1
        Return $FindSched(M, n-1)$
```

**Running time:** $O(n)$. # of recursive calls $= n - 1$ decrements down to base case $= O(n)$, and each call takes constant time.

**Example:**
$FS(M,6) : M[5] vs. M[3] + v_6 . 8 vs. 6 + 1 =$ exclude 6.
$FS(M,5) : M[4] vs. M[3] + v_5 . 7 vs. 6 + 2 =$ include 5.
$FS(M,3) : M[2] vs. M[1] + v_3 . 4 vs. 2 + 4 =$ include 3.
$FS(M,1) : 1.$
Schedule: $\{5,3,1\}$.

**Space complexity:**

# Chapter 9

# Knapsack

Subproblems have more than 1 variable.

Knapsack function

**Input:** $n$ items (of value $v_i$, weight $w_i \in \mathbb{N}$) for your knapsack (capacity $T \in \mathbb{N}$).
**Problem:** Optimally choose items to get the greatest value that can fit in the knapsack.
**Output:** The most valuable subset of items ($S \subseteq \{1, ..., n\}$) that fit in the knapsack: $argmax_{S\subseteq\{1,...,n\}} V_s$ such that $W_s \leq T$.
Maximize value $V_s = \sum_{i \in S} v_i$.
Weight $W_s = \sum_{i \in S} w_i$ at most $T$.

EXTRA: Other versions of the knapsack problem:

- SubsetSum: $v_i = w_i$.
- TugOfWar: $v_i = w_i$, $T = \frac{1}{2} \sum_i v_i$.

## 9.1 Why DP is needed

It seems we should pick items that are very valuable with little weight (have a large $\frac{v_i}{w_i}$). This strategy is greedy. This strategy is not optimal.

**Example:** $T = 8, (v_1 = 6, w_1 = 5), (v_2 = 4, w_2 = 4), (v_3 = 4, w_3 = 4)$.
Ratios: $\frac{v_1}{w_1} = \frac{6}{5}$, $\frac{v_2}{w_2} = \frac{4}{4} = 1$, $\frac{v_3}{w_3} = \frac{4}{4} = 1$.
If $T = 8$, we should pick $\{1\}$ and we can't fit any more items in. $V = 6$.
This solution is not optimal, $\{2, 3\}$ is better: has value $= 8$, and weight $= 8 \leq$ the max weight, $T = 8$.

## 9.2 The $n$-th item

**Want:** $argmax_{S\subseteq\{1,...,n\}} V_S$ such that $W_S \leq T$.

- Case 1: $n$-th item is in optimal solution.
- Case 2: $n$-th item is not in optimal solution.

## 9.3 Subproblems

Let $O_n \subseteq \{1, ..., n\}$ be the optimal subset of items, given the first $n$ items.

- Case 1: $n \notin O_n$. Then $O_n = O_{n-1}$ with same capacity.
- Case 2: $n \in O_n$. Then $O_n = \{n\} \cup O_{n-1}$ with capacity = previous capacity - $w_n$.

2 variables: value (optimized), and capacity left

## 9.4 Recurrence

Let $OPT(j, S)$ be the value of the optimal subset of items {1, ..., $j$} in a knapsack of size $S$.

- Case 1: $j \notin O_{j,S}$. Then $OPT(j, S) = OPT(j-1, S)$, the optimal value of first $j-1$ items with capacity $S$.
- Case 2: $j \in O_{j,S}$. Then $OPT(j, S) = v_j OPT(j-1, S-w_j)$, the value of $j$ + optimal value of first $j-1 items$ with capacity reduced by $j$'s weight.

**Recurrence:**

$$OPT(j,S) = \begin{cases} max(OPT(j-1,S), v_j + OPT(j-1, S-w_j)), & w_j \leq A. \quad \text{Maximize value: choose optimal case.} \\ OPT(j-1,S), & w_j \leq S. \quad \text{Only the first case.} \end{cases}$$

**Base cases:**
$OPT(j, 0) = 0$. Can't fit any items into capacity 0.
$OPT(0, S) = 0$. No items to put in knapsack.

## 9.5 Fill in table "Bottom Up"

Knapsack Bottom Up

```
▷ All inputs are global variables. ◁
function FindOPT(n, T)
    ▷ n = number of items ◁
    ▷ T = capacity of knapsack ◁
    for j = 1, ..., n do                          ▷ Label items in arbitrary order.
        M[j, 0] ← 0                                ▷ Base case
        for S = 1, ..., T do
            M[0, S] ← 0                            ▷ Base case

            if $w_j > S$ then
                $M[j, S] ← M[j-1, S]$
            else
                $M[j, S] ← max\{M[j-1, S], v_j + M[j-1, S-w_j]\}$

    return $M[n, T]$
```

Iterates over every possible item and every possible capacity.
**Running time:** $O(nT)$. DP table is $n$ by $T = nT$ iterations, and each iteration (filling in an entry) takes constant number of constant time operations = $O(1)$.
**Space complexity:** $O(nT)$. DP table is $n+1$ by $n+1$ (+1 accounts for 0's). There are $n$ items, and each item has constant number of attributes to store (optimal value) = $O(1)$.

### 9.5.1 Example: Fill in table

**Input:**

- $T = 8, n = 3$
- $w_1 = 2, v_1 = 4$
- $w_2 = 3, v_2 = 5$
- $w_3 = 5, v_3 = 8$

**Table:**

first $j$ items

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 3 | 0 | 0 | 4 | 5 | 5 | 9 | 9 | 12 | 13 |
| 2 | 0 | 0 | 4 | 5 | 5 | 9 | 9 | 9 | 9 |
| 1 | 0 | 0 | 4 | 4 | 4 | 4 | 4 | 4 | 4 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| - | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

capacities $S$

**Filling in the table:**
(1, 1): Item 1 cannot fit in capacity 1 (case 2) = OPT(0, 1) = 0.
(1, 2): Item 1 fits in capacity 2 (case 1) = max(OPT(0,2), 4 + OPT(0,0)) = max(0, 4 + 0) = 4.
(1, 3): Item 1 fits in capacity 3 (case 1) = max(OPT(0, 3), 4 + OPT(0, 1)) = max(0, 4 + 0) = 4.
...
(2, 1): Item 2 cannot fit in capacity 1 (case 2) = OPT(1,1) = 0.
...
(3, 7): Item 3 fits in capacity 7 (case 1) = max(OPT(2,7), 8 + OPT(2, 3)) = max(9, 8 + 4) = 12.

## 9.6 Recovering solution from table

Let $O_{j,S}$ be the optimal subset of items {1,...,$j$} in a knapsack of size $S$.

- Case 1: $j \notin O_{j,S}$. Get optimal solution for items 1 to $j - 1$ in a knapsack size of $S$.
- Case 2: $j \in O_{j,S}$. Get $j$ + optimal solution for items 1 to $j - 1$ in a knapsack size of $S - w_j$.

**Knapsack: Recovering solution**

```
▷ All inputs are global vars ◁
▷ M[0:n, 0:T] contains solutions to subproblems. ◁
function FindSol(M,n,T)
    if $n = 0 or T = 0$ then
        return $\emptyset$
    else
        if $w_n > T$ then
            return $FindSol(M, n-1, T)$
        else
            if $M[n-1, T] > v_n + M[n-1, T-w_n]$ then
                return $FindSol(M, n-1, T)$
            else
                return $\{n\} + FindSol(M, n-1, T-w_n)$
```

**Running time:** $O(n)$. In the worst case, look at each item (each call decrements $n$ by 1 each time).

# Chapter 10

# Segmented Least Squares

Recurrence has multi-way base cases.

## 10.1 Background: Least Squares

Least Squares function

**Input:** $n$ data points $P = \{(x_y, y_1), ..., (x_n, y_n)\}$
**Problem:** Find the line of best fit. **Output:** The line $L$ ($y = ax + b$) that fits best.

**Equations:**

- **best:** = minimizes $error(L, P) = \sum_i (y_i - ax_i - b)^2$.
- **slope:** $a = \frac{n\sum(x_i y_i) - (\sum x_i)(\sum y_i)}{n \sum x_i^2 - (\sum x_i)^2}$.
- **y-intercept:** $b = \frac{\sum y_i - a \sum x_i}{n}$.

**Running time:** Finding the line of best fit takes $O(n)$ (summing $n$ values).

## 10.2 Segmented Least Squares: Version 1

When the data points are not in a line, use multiple lines of best fit.
**Segment:** A group of points.
**Line:** Line of best fit on that group.
Using $n/2$ segments is pointless (a line between each pair of points doesn't describe overall data well).

**Version 1:** Solve SLS with a "segment cost" in time $O(n^2)$ and space $O(n^2)$. Uses multiway case analysis for the final segment (instead of 1 case to consider, there are MANY cases to consider).

**Segmented Least Squares function**

**Input:** $n$ data points $P = \{(x_y, y_1), ..., (x_n, y_n)\}$ where points are labeled by $x$ coordinate: $x_1 < x_2 < ... < x_n$, cost parameter $C > 0$.
**Problem:** Find the lines of best fit that minimizes error and cost. **Output:** A partition of $P$ into contiguous (disjoint) segments $S_1, S_2, ..., S_m$), lines $L_1, L_2, ..., L_m$, minimizing cost.

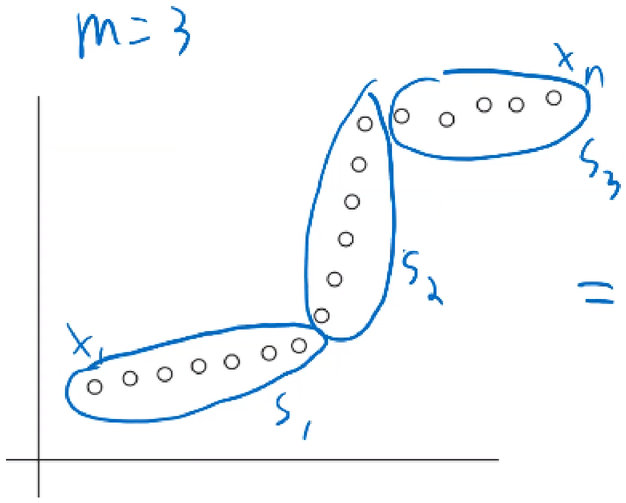
$cost(S_1, ..., S_m, L_1, ..., L_m) = mC + \sum_{i=1}^{m} error(L_i, S_i)$

**Observations:**

- Every time you include a new segment, you must pay the cost parameter $\longrightarrow$ Discourages making new segments.
- **partition:** includes all points.
- **contiguous:** consecutive points in segments, can't skip over a point.
- **disjoint:** segments don't overlap.
- For every segment $S_j$, $L_j$ must be the single line of best fit for $S_j$.
  - Let $L*_{i,j}$ be the optimal line for $\{p_i, ..., p_j\}$.
  - Let $\mathcal{E}_{i,j} = error(L*_{i,j}, \{p_i, ..., p_j\})$, the error between line of best fit and segment.
  - Can compute $\mathcal{E}_{i,j}$ for all $(i, j)$ in $O(n^3)$ time straightforwardly ($\binom{n}{2} = O(n^2)$ to choose a start and end point * $O(n)$ to find line of best fit $= O(n^3)$), or $\underline{O(n^2)}$ time (by using the previous error term to calculate the next error term).

### 10.2.1 Segmentation Example

If $m = 3$:
**Segmentation:**



**Cost:** $cost = 3C + error(L_1, S_1) + error(L_2, S_2) + error(L_3, S_3)$.

### 10.2.2 SLS Example

**Input:** $\{A = (1,1), B = (2,1), C = (3,3)\}$.

| Potential segment | Optimal line | Error |
|---|---|---|
| $[A]$ | $y = 1$ | 0 |
| $[B]$ | $y = 1$ | 0 |
| $[C]$ | $y = 3$ | 0 |
| $[A, B]$ | $y = 1$ | 0 |
| $[B, C]$ | $y = 2x - 3$ | 0 |
| $[A, B, C]$ | $y = x - 1/3$ | 2/3 |

**Possible segmentations:**

1. ~~$\{[A], [B], [C]\}$ has cost $= 0 + 0 + 0 + 3C$.~~
2. $\{[A, B], [C]\} = \{[A], [B, C]\}$ have cost $= 0 + 0 + 2C = 2C$. $m = 2$. These also have error of 0 but cost less than $3C$.
3. $\{[A, B, C]\}$ has cost $= 1C + 2/3$. $m = 1$.

$$\begin{cases} \text{if } C > 2/3, & \text{Partition 3 is optimal.} \\ \text{if } C < 2/3, & \text{Partition 2 is optimal.} \end{cases}$$

**Choosing C:** When choosing $C$, If you want fewer segments, decrease $C$. If you're okay with more segments, increase $C$.

## 10.3 Finding the Recurrence

**Last segment:**
Let $O_j$ be the optimal solution for $\{p_1, ..., p_j\}$. Then $p_j$ is the last point in the last segment in $O_j$: $[p_i, ..., p_j]$ for $1leqi \leq j$.
If the final segment is $[p_i, ..., p_n]$, then the optimal solution for $\{p_1, ..., p_n\}$ is $O(n) = [p_i, ..., p_n]$ (the final segment) $\cup O_{i-1}$ (optimal solution for first $p_1$ to $p_{i-1}$ points).
There are a lot of possible choices for the first point in the last segment! We need to consider all of these cases...

**Multi-way Choices:**
Let $OPT(j)$ be the **value** (total cost) of the optimal solution for points $\{p_1, ..., p_j\}$ (the first $j$ points).

- **Case $i$:** final segment is $\{p_i, ..., p_j\}$ (final segment starts with $p_i$). Optimal solution $= L*_{i,j} \cup OPT(i-1)$ (optimal solution for $O_{i-1} = \{p_1, ..., p_{i-1}\}$). If we fix $j$, there are $j$ possible values for $i$: $i \in \{1, ..., j\}$.

**Total cost:** 1 + 2 + 3

1. $\mathcal{E}_{i,j}$
2. $C$ (cost parameter), to pay for segment $\{p_i, ..., p_j\}$.
3. $OP(i - 1)$, the cost for the rest of the points (recursive).

## 10.4 Recurrence

**Recurrence:** $OPT(j) = \min_{1 \leq i \leq j} (\mathcal{E}_{i,j} + C + OPT(i - 1))$. Find the minimum cost, over ALL different values for $i$ (multi-way cases).

**Base cases:**
$OPT(0) = 0$. Segment with 0 points = 0 cost.
$OPT(1) = OPT(2) = C$. Segments with 1 or 2 points = pay for 1 segment.

## 10.5 SLS without DP

SLS without DP

```
▷ All inputs are global vars ◁
function FINDOPT(n)
    if $n = 0$ then
        return 0
    else if $n = 1, 2$ then
        return $C$
    else
        return $\min_{1 \le i \le j} (\mathcal{E}_{i,j} + C + \text{FINDOPT}(i-1))$
```

**Running time:** exponential or worse... no memoization & keeps recomputing solutions.

## 10.6 SLS "Top Down"

Calculate costs and fill in table.

SLS Top Down

```
▷ All inputs are global vars ◁
$M \leftarrow$ empty array, $M[0] \leftarrow 0$, $M[1] \leftarrow C$, $M[2] \leftarrow C$        ▷ Store bases in DP table.
function FINDOPT(n)
    if $M[n]$ is not empty then
        return $M[n]$        ▷ Already computed.
    else
        ▷ Computer for the first time: use recurrence. ◁
        ▷ Error already calculated in pre-processing step. ◁
        ▷ Over all values of $j$: 1, 2, 3, ..., n. Could represent as a FOR loop. ◁
        $M[n] \leftarrow \min_{1 \le i \le j} (\mathcal{E}_{i,j} + C + \text{FINDOPT}(i-1))$
        return $M[n]$
```

**Running time:**
Filling in $n - 2 = O(n)$ elements (exclude bases $n = 1$ and $n = 2$).
To fill $M[j]$ (for each value of $j$) we make $j = O(n)$ recursive calls ($i$ iterates from 1 to $n$. Or, there are $j$ different values of $i$ and is upper-bounded by $n$).
Total number of recursive calls: $\sum_{j=3}^{n} j = O(n^2)$. Sums over all possible values of $j$: 3 recursive calls + 4 recursive calls + 5 recursive calls ... + $n$ recursive calls.
Total run time: $O(n^2)$. There are $O(n^2)$ recursive calls, and each call takes constant time. Remember, we assumed we already calculated the errors, which can be done in $O(n^2)$ with cleverness.

## 10.7 SLS "Bottom Up"

Calculate costs and fill in table.

SLS Bottom Up

```
▷ All inputs are global vars ◁
function FindOPT(n)
    $M \leftarrow$ empty array, $M[0] \leftarrow 0$, $M[1] \leftarrow C$, $M[2] \leftarrow C$    ▷ Store bases in DP table.
    for $= 3, ..., n$ do
        ▷ Can represent min as nested FOR loop. ◁
        $M[n] \leftarrow \min_{1 \leq i \leq j} (\mathcal{E}_{i,j} + C + \text{FindOpt}(i-1))$
    return $M[n]$
```

In DP table, we always look up smaller values = DP is always already filled.
**Running time:** $O(n^2)$. Nested *For* loop: $j = 3, ..., n = O(n)$ loops * $i = 1, ..., j = O(n)$ ($j$ could be $n$) = $O(n^2)$.

### 10.7.1 Example: Bottom Up

**Input:** $A = (1,1)$, $B = (2,1)$, $C = (3,3)$.
**Bases:** $M[0] = 0$, $M[1] = C$, $M[2] = C$.
For $j = 3$, these are the costs for different starting point $i$:

- $i = 1$: cost $= C + \mathcal{E}_{1,3} + OPT(1-1)$ (segment A,B,C) $= C + 2/3 + 0 = C + 2/3$.
- $i = 2$: cost $= C + \mathcal{E}_{2,3} + OPT(2-1)$ (segment B,C) $= C + 0 + C = 2C$.
- $i = 3$: cost $= C + \mathcal{E}_{3,3} + OPT(3-1)$ (segment C) $= C + 0 + C = 2C$.

$M[3] = min(C + 2/3, 2C, 2C)$.

## 10.8 Recovering solution

Remember, $O_j$ is the optimal solution for $\{p_1, ..., p_j\}$. $OPT(j)$ is the total cost of the optimal solution for points $\{p_1, ..., p_j\}$ (the first $j$ points).

If $x == argmin_{1 \leq \leq n}(\mathcal{E}_{i,n} + C + M[i-1])$, then $O_{x-1} \cup [p_x, ..., p_n]$.

- $x =$ the $i$ that minimizes $(\mathcal{E}_{i,n} + C + M[i-1])$.
- $x$ is a value of $i$ (index of 1st point in last segment: $[p_x, ..., p_n]$).

Finding Segments

```
▷ All inputs are global vars ◁
▷ M[0:n] contains solutions to subproblems (already filled). ◁
function FindSol(M,n)
    if $n = 0$ then
        return $\emptyset$
    else if $n = 1$ then
        return $\{1\}$
    else if $n = 1$ then
        return $\{1, 2\}$
    else
        Let $x \leftarrow argmin_{1 \leq \leq n}(\mathcal{E}_{i,n} + C + M[i-1])$
        return $\{x, ..., n\} +$ FindSol$(M, x-1)$
```

**Running time:** $O(n^2)$. Argmin takes $O(n)$: loops over $n$ different values of $i$ and checks "is this the min so far?" using the table. Each loop takes $O(n)$: looks up error, looks up $M[i-1]$, adds the 3 pieces together. Number of calls/segments $= O(n)$: Number of segments depends on $C$, but cannot be more than $n/2$ in the worst case.

**Space complexity:** $O(n^2)$. DP table is $n+1$ by 1 array $= O(n)$. Storing error terms ($\mathcal{E}_{i,j}$'s) $= O(n^2)$: each pair of points $(i, j)$ corresponds to a segment, so there are $\binom{n}{2} = O(n^2)$ possible segments. Storing each point $= O(n)$. The dominant term is storing the error terms, so total $= O(n^2)$.

## 10.9 EXTRA: Segmented Least Squares: Version 2

Instead of a cost parameter, an upper bound on the number of segments allowed ($k$) is given.

Segmented Least Squares function

**Input:** $n$ data points $P = \{(x_y, y_1), ..., (x_n, y_n)\}$, parameter $1 \leq k \leq n$: a hard upper bound on the number of segments .
**Problem:** Find the lines of best fit that minimizes error within number of segments allowed.
**Output:** A partition of $P$ into $\leq k$ contiguous (disjoint) segments $S_1, S_2, ..., S_k$), minimizing "cost".

$cost(S_1, ..., S_m, L_1, ..., L_m) = \sum_{i=1}^{k} error(L_i, S_i)$

This is an example of a parameterized algorithm.

# Part II

# Proof techniques

# Chapter 11

# Proof by Induction

Used to prove a claim $H$ is true for every natural number $i$ starting at a first value (usually 0 or 1). $H(i)$ is true $\forall i$.
Steps:

1. Base case: prove directly (by plugging in the base cases).
2. Inductive step: for a general $k \in \mathbb{N}$, show $H(k-1) \implies H(k)$. Use the **inductive hypothesis (IH)**: assume $H(k-1)$ is true. Alternatively, show $H(k) \implies H(k+1)$ and assume $H(k)$ is true.
    - Strong induction: We may assume $H(1), H(2), \ldots\ H(i-1)$ to prove $H(i)$ Some number in this chain will be helpful (not just $i-1$). Cases may be helpful, but not necessary.
    - Weak induction: We only need to assume $H(i-1)$ because it is all we need to prove $H(i)$. Anything proven with weak induction can be proven with strong induction. Weak induction is like having all previous steps proven, but only using the $(i-1)$ step.

Explanation:

- Starting from the base, $H(1) \implies H(2)$ by inductive step $\implies H(3) \implies \ldots \implies H(100) \implies \ldots$, to any natural number!

Note that induction can be used to prove algorithms, and does not only apply to the following examples.

### 11.0.1 Summation

**Claim:** For every $n \geq 1, \sum_{i=0}^{n-1} 2^i = 2^n - 1$.

*Proof by Induction.* **Base:** $n = 1$. LHS: $\sum_{i=0}^{i=0} 2^i = 2^0 = 1$. RHS: $2^1 - 1 = 1$. LHS = RHS. $H(1)$ is true.

**Inductive step:** Assume $H(n)$ holds. Show $H(n+1)$, that $\sum_{i=0}^{n+1-1} 2^i = 2^{n+1} - 1$.

$$\begin{aligned}
&\sum_{i=0}^{n+1-1} 2^i \\
&= \sum_{i=0}^{i=n} 2^i \\
&= 2^0 + 2^1 + 2^2 + \ldots + 2^n && \text{The } n-1 \text{ terms are in IH.} \\
&= 2^n - 1 + 2^n && \text{By induction hypothesis.} \\
&= 2 * 2^n - 1 \\
&= 2^{n+1} - 1
\end{aligned}$$

**Alternatively,** $H(k-1) \implies H(k)$.
Assume $\sum_{i=0}^{i=k-1-1} 2^i = 2^{k-1} - 1 = \sum_{i=0}^{i=k-2} 2^i = 2^{k-1} - 1$ is true. Show $H(k) : \sum_{i=0}^{i=k-1} 2^i = 2^n - 1$.
Pull the last term out of the summation.

$$\begin{aligned}
&\sum_{i=0}^{i=k-2} 2^i + \sum_{i=k-1}^{i=k-1} 2^i \\
&= \textstyle\sum_{i=0}^{i=k-2} 2^i + 2^{k-1} \\
&= 2^{k-1} - 1 + 2^{k-1} && \text{By induction hypothesis.} \\
&= 2 * 2^{k-1} - 1 \\
&= 2^k - 1
\end{aligned}$$

□

### 11.0.2 Prime factors

**Claim:** $\forall n \in \mathbb{N}, n \geq 2$ has at least 1 prime factor.

**prime** Positive integer greater than 1 that cannot be formed by multiplying two smaller natural numbers.

**factor** A number that divides another number evenly.

We need to use strong induction. Weak induction won't work: 11 having a prime factor doesn't help determine if 12 has a prime factor.

*Proof by Induction with Cases.* **Base:** 2 is a natural number that has the prime factor 2.
**Inductive step:** Assume $H(2), H(3), \ldots, H(k-1)$ are all true. Show $H(k)$ is true, that $k$ has a prime factor.
Proof by cases:

- Case 1: $k$ is prime. $k$ is a factor of itself ($k * 1 = k$), so $k$ is a prime factor of $k$.
- Case 2: $k$ is not prime, so $k$ is divisible by a natural number less than $k$. $k = a * b$ where $a, b \in \mathbb{N}$ and $2 \leq a, b < k$. By IH, all natural numbers less than $k$ have a prime factor, so $H(a)$ is true (we don't know where $a$ is in the chain, but it's somewhere in there). Therefore, $a$ has a prime factor: $a = x * y$ where $x$ is prime. $k = a * b = x * y * b$, where $x$ is prime and $y * b$ is a natural number. Therefore, $x$ must be a prime factor of $k$.

□

### 11.0.3 Inequalities

**Claim:** $\forall n \geq 4, n^2 \leq 2^n$.

*Proof by Induction.* **Base:** $n = 4$. LHS: $4^2 = 16$. RHS $= 2^4 = 16$. LHS $\leq$ RHS. The base case for $n = 4$ holds true.
**Inductive step:** Assume that $k^2 \leq 2^k$ for some $k \geq 4$. Show $(k+1)^2 \leq 2^{k+1}$.

$$\begin{aligned}
&(k+1)^2 && \\
&= k^2 + 2k + 1 && \\
&\leq 2^k + 2k + 1 && \text{By induction hypothesis} \\
&\leq 2^k + 2^k && \text{since } 2k+1 \leq 2^k, \forall n \geq 4 \text{ (see subproof)} \\
&= 2^{k+1}. &&
\end{aligned}$$

□

*Subproof by Induction.* **Claim:** $2n + 1 \leq 2^n, \forall n \geq 4$
**Base:** $n = 4$. LHS $= 2(4) + 1 = 9$. RHS $= 2^4 = 16$. LHS $\leq$ RHS. The base case for $n = 4$ holds true.
**Inductive step:** Assume that $2k + 1 \leq 2^k$ for some $k \geq 4$. Show $2(k+1) + 1 \leq 2^{k+1}$.

$$\begin{aligned}
&2(k+1) + 1 && \\
&= 2k + 2 + 1 && \\
&= 2k + 1 + 2 && \\
&\leq 2^k + 2 && \text{By induction hypothesis} \\
&\leq 2^k + 2^k && \text{since } 2 < 2^k, \forall k \geq 4 \\
&= 2(2^k) && \\
&= 2^{k+1}. &&
\end{aligned}$$

□

Chapter 12

# Proof by Contradiction

Explanation:

- No claim can be both true and false.

Steps:

1. Assume the claim $H$ is false ($\neg H$ is true).
2. Show $H$ being false implies contradictory assertions (that both assertion $Q$ and $\neg Q$ are true)
3. $Q$ and $\neg Q$ cannot both be true, so $H$ must not be false ($H$ is true).

# Part III

# Algorithm techniques

# Chapter 13

# Divide & Conquer

Divide a big problem into smaller instances of the same problem. Recursively solve each subproblem, repeating until we reach "small enough" instances that can be "conquered". Combine solutions to the subproblems to solve the original big problem.
Tools:

- Prove correctness: proof by induction
- Analyze runtime: recurrences & recursion trees
- Asymptotic Analysis

### 13.0.1 Mergesort: sorting a list

Click here.

### 13.0.2 Karatsuba's Algorithm: integer multiplication

Click here.

### 13.0.3 Finding the $k^{th}$ largest element in an array

Click here.

## 13.1 Reduce and Conquer

Throw away part of the problem, and work on just 1 new smaller instance. Don't need to combine solutions later.

### 13.1.1 Binary Search: search in a sorted list

Click here.

Chapter 14

# Dynamic Programming (DP)

- Like Divide and Conquer, breaks the problem up into smaller pieces and recursively solves.
- Unlike D&C, **reuses solutions as necessary when subproblems repeat, instead of re-solving problems.**
- Don't need to combine solutions. Subproblems are standalone but feed into the larger problem.
- Often the only polynomial algorithm (D&C doesn't work)
- Usually used for computing a value that is optimal based on an objective.

## 14.1 Dynamic Programming Recipe

**Recipe:**

1. Identity a set of **subproblems**.
2. Relate the optimal solution on subproblems as a **recurrence**.
3. Find an **efficient implementation** of the recurrence (top down or bottom up). Solve for the value of the optimum.
    - Top-down: recursive, store solutions to subproblems.
    - Bottom-Up: iterate through subproblems in order.
4. **Reconstruct the solution** from the DP table of values.

## 14.2 DP tips

**Recreating the solution should not dominate the running time** in general.

**DP Space complexity depends on how large DP table is.** Larger space complexity is a drawback of DP.

**Each variable adds 1 dimension** in the DP table.

- Iterate over increasing values. As long as we look up entries smaller, they will be computed already.
- Both methods fill in the table from smaller subproblems $\longrightarrow$ larger subproblems.

## 14.3 "Top Down"

- Recursive.
- Starts with the big value, and leads to subsequent recursive calls.
- Running time usually depends on # recursive calls.
- Only solve problems you actually need to use because they are directly following from the recurrence = DP table has blank entries for unused subproblems.

## 14.4 "Bottom Up"

- Iterative instead of recursive. Usually has same time complexity as Top Down.
- Solve the small problems, then build up to the big problem.
- Running time usually depends on # loops.
- Start from high values to low values of $n$.
- Fills the DP entire table, even subproblems that won't be used.

## 14.5 Fibonacci Numbers

The $n^{th}$ Fibonacci number = sum of the 2 previous numbers.

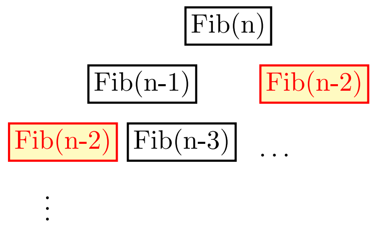
- 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...
- Base cases: $F(1) = 0, F(2) = 1$
- Recursive: $F(n) = F(n-1) + F(n-2)$
- Recursive running time: $F(n) = \Theta(\phi^n) \approx \Theta(1.62^n)$, the golden ratio: $\phi = \left(\frac{1+\sqrt{5}}{2}\right)$. This recomputes the same values $F(i)$ many times.
- To improve running time...
  1. Top down: remember values already computed.
  2. Bottom up: Iterate over all values $F(i)$.
- Fastest algorithms solve in logarithmic time.

### 14.5.1 Fibonacci Numbers: Version 1 without DP

**Fibonacci Numbers: without DP**

```
function FibI(n)
    if $n = 1$ then
        Return 0
    else if $n = 2$ then
        Return 1
    else
        Return $FibI(n-1) + FibI(n-2)$                ▷ Recurse
```

**Running time:** $T(n) = \#$ calls by `FibI(n)`.



**Bases:** $T(1) = T(2) = 1$.
**Recurrence:** $T(n) = T(n-1) + T(n-2) + 1$, left branch + right branch + initial `FibI(n)` call. This is the same as $T(n) = 2F(n+1) - 1$, from the pattern below. Results in a lot of recursive calls, because $F(n)$ is exponential: $\approx 2 * 1.62^{n+1} - 1 = O(1.62^n)$
This algorithm is correct but slow. It doesn't reuse solutions (calculates F(n-2) twice). We want to store F(n-2) once its calculated so we can use it again.

| $i$ | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| $F(n)$ | 0 | 1 | 1 | 2 | 3 | 5 |
| $T(n)$ | 1 | 1 | 3 | 5 | 9 | 15 |

### 14.5.2 Fibonacci Numbers: Version 2 ("Top Down" DP)

Create a DP table of size $n$ to store solutions.

**memoization** Storing solution to sub-problems.

**Fibonacci Numbers: Top Down DP**

```
$M \leftarrow$ empty array, $M[1] \leftarrow 0$, $m[2] \leftarrow 1$      ▷ To store solutions. Add base cases
function FibII(n)
    if $M[n]$ is not empty then
        Return $M[n]$                                    ▷ Reusing solutions
    else if $M[n]$ is empty then
        ▷ If we haven't computed it yet, use the recurrence ◁
        $M[n] \leftarrow FibII(n-1) + FibII(n-2)$
        Return $M[n]$
```

**Running time:** Let $T(n) = \#$ calls to `FibII(n)`. There are $n - 2$ elements to fill in DP table (excludes 2 base cases). Each fill uses 2 recursive calls (even if the subproblem is already calculated, we still need to look it up. Solved solutions don't trigger more recursive calls though). Total: $2(n-2) = O(n)$.
**Example:** To calculate $n = 4$:

$M =$

| | | | |
|---|---|---|---|
| 0 | 1 | | |

Fib(4): M[4] is empty $\longrightarrow$ Fib(3) + Fib(2)
Fib(3): M[3] is empty $\longrightarrow$ Fib(2) + Fib(1)
Fib(2) and Fib(1) are in the table $\longrightarrow$ 1 + 0
Fill in M[3] = Fib(3)

### 14.5.3 Fibonacci Numbers: Version 3 (”Bottom Up” DP)

In both top down and bottom up, you fill up the table in the same order. But top down starts with a recursive call, whereas bottom up starts with filling in the table for small values of $i$.

**Fibonacci Numbers: Bottom Up DP**

```
1: $M \leftarrow$ empty array, $M[1] \leftarrow 0$, $m[2] \leftarrow 1$        ▷ To store solutions. Add base cases
2: function FibIII(n)
3:     for $i = 3, ..., n$ do
4:         Return $M[i] \leftarrow FibIII(i-1) + FibIII(i-2)$
5:     Return $M[n]$
```

**Running time:** Let $T(n) = \#$ calls to `FibII(n)`. There are $n-2$ iterations in the *For* loop, each iteration take $O(1)$ time (look up and addition is constant time). Total: $O(n)$.

### 14.5.4 Following the DP recipe:

1. Identity a set of **subproblems**: $F(i)$, the $i^{th}$ Fibonacci number.
2. Relate the subproblems via a **recurrence**: $F(i) = F(i-1) + F(i-2)$
3. Find an **efficient implementation** of the recurrence (top down or bottom up).
4. **Reconstruct the solution** from the DP table.

## 14.6 Weighted Interval Scheduling

Click here.

## 14.7 Knapsack

Click here.

## 14.8 Segmented Least Squares

Click here.

# Part IV

# Analysis

Predicting the wall-clock time of an algorithm is nearly impossible.

- Different machines can have more computational power.
- Time per operation fluctuates.

# Chapter 15

# Asymptotic Analysis

How does running time grow as the size of the input grows?

- Tool used to compare algorithms by asymptotic order of growth, the behavior as $n \implies \infty$. # operations $\rightarrow$ runtimes $\rightarrow$ functions. Order of growth functions are easier to work with than the exact running time/# of operations. Can also be used to describe space complexity, # of nodes/edges in a graph, or compare parameters.
- Describes performance based on input size $n$.
- Measures speed/size as input grows very big to see how algorithm scales. Focuses on the dominant (largest) term of function. Generally we don't care about small inputs, because then all algorithms will be reasonably quick.
- Compare order of growth for algorithms that scale well. Differences in order of growth have HUGE effects when $n$ is very large.

## 15.1 Finding Running Time (Counting Operations)

The following counts as 1 operation:

- Creating (and assigning a value to) a variable
- Assigning a value to an existing variable
- Getting the value of a variable
- Arithmetic (+, -, *, /, %)
- Comparisons ($<, >, =, \leq, \geq$)
- Function calls (1 function call + # of operations inside the function)

**Note:** To find the runtime of an algorithm, don't need to find constants for formal proof unless asked to show an algorithm is Big-Oh of a function. Count the operations, then drop the lower order terms and constants. Note that you cannot drop variables, unless a specific constant value is specified for them. Example.
Give tight bounds. Looser bounds are true, but not meaningful. Tight runtime bounds give meaningful description of algorithm.
**Note:** Note: $C$ from base case is not the same $C$ as in $T(n)$. Asymptotically doesn't matter though, they are all constants.

## 15.2 "Big-Oh" Notation: Worst case

"Big-Oh" Notation

$f(n) = O(g(n))$ if there exists $c \in (0, \infty)$ and $n_0 \in \mathbb{N}$ such that $f(n) \leq c * g(n)$ for every $n \geq n_0$.

- $f(n)$ is exact run time based on # of operations ("messy"). $g(n)$ is order of growth ("nice").
- Asymptotic version of $f(n) \leq g(n)$.
- Roughly equivalent to $\lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$. Denominator grows faster = converges. Can solve using L'Hopitals.
- **There are many combos of $c$ and $n_0$ that will work in the inequality.**
- $f(n)$ can be Big Oh of many functions, starting from the order of growth.
- **All $\log_x n$ functions are Big-Oh of each other.** Asymptotically equivalent because changing base is multiplying by a constant.
- If $f(n) = O(g(n))$, $g(n)$ can be infinitely many functions.
- Used most often (simpler, worst case behavior). Should be as precise/tight as possible.

### 15.2.1 An easy way to find $c$ and $n_0$

1. To get $c$, sum the coefficients in $f(n)$.
2. Usually $n_0 = 1$. $n_0$ is the size at which the algorithm's $n \to \infty$ behavior starts. In log vs. $n^2$, log starts outcompeting $n^2$ at $n_0 = 160$. $n_0$ is any point where the functions cross each other or later. Try to pick a low $n_0$, really really big $n_0$ are valid but not meaningful.

### 15.2.2 Proving a function is Big-Oh of another function

Find values for $c$ and $n_0$ such that the definition for Big-Oh holds.
**Prove:** $3n^2 + n = O(n^2)$ is true.

*Proof.* $f(n) = 3n^2 + n$. $g(n) = n^2$. Find constants such that $3n^2 + n \leq cn^2, \forall n \geq n_0$.
Sum the coefficients: $c = 3 + 1 = 4$.
Choose an easy number: $n_0 = 1$.
Show that the values work:

$$\begin{aligned} 3n^2 + n &\leq 4n^2 && \forall n \geq 1 \\ n &\leq n^2 && \forall n \geq 1 \\ 1 &\leq n && \forall n \geq 1. \text{ This is true, so the bound holds true.} \end{aligned}$$

There exist constants $c$ and $n_0$, so $3n^2 + n = O(n^2)$ is true. □

#### More Examples

- $n^3 \neq O(n^2)$. $\lim_{n\to\infty} f(n)/g(n) = \frac{n^3}{n^2} = \infty$, limit diverges.
- $10n^4 = O(n^5)$. Let $c = 10, n_0 = 1$. $10n^4 \leq 10n^5, \forall n \geq 1$. $1 \leq n, \forall n \geq 1$.
- $\log_2 n = O(\log_{16} n)$. Change of base: $\log_2 n = \frac{\log_{16} n}{\log_{16} 2} = \frac{\log_{16} n}{\frac{1}{4}} = 4\log_{16} n$. Let $c = 4, n_0 = 1$. $4\log_{16} n \leq 4\log_{16} n$ for any value of $n_0$ because they are equal!

### 15.2.3 $f(n) + g(n) = O(h(n))$

**Claim:** If $f(n) = O(h(n))$ and $g(n) = O(h(n))$, then $f(n) + g(n) = O(h(n))$.

*Proof.* If $f(n) = O(h(n))$, then $f(n) \leq c * h(n), \forall n \geq n_0$ for some constants $c, n_0$.
If $g(n) = O(h(n))$, then $g(n) \leq c' * h(n), \forall n \geq n_0'$ for some constants $c', n_0'$.

Show $f(n) + g(n) \leq c'' * h(n), \forall n \geq n0''$, for some constants $c'', n_0''$.

$$\begin{aligned}
&f(n) + g(n) \\
&\leq c * h(n) + c' * h(n) \\
&= (c + c') * h(n), \qquad \forall n \geq max(n_0, n_0').
\end{aligned}$$

$\therefore c'' = c + c', n_0'' = max(n_0, n_0')$.
Alternatively, $n_0'' = n_0 + n_0'$ because the sum is $\geq n_0$ and $n_0'$. $\square$

### 15.2.4 Asymptotic Analysis Rules

Computing order of growth:

- **Constant factors can be ignored.** $\forall C > 0, Cn = O(n)$.
- **Lower order terms can be dropped.** $n^2 + n^{3/2} + n = O(n^2)$.

Comparing/Ranking orders of growth:

- **Smaller exponents are Big-Oh of larger exponents.** $\forall a > b, n^b = O(n^a)$. For instance, $n^2 = O(n^{2.000000001})$.
- **Any logarithm is Big-Oh of any polynomial.** $\forall a, b > 0, \log_2^a n = O(n^b)$. For instance, $(\log_2 n)^{100000} = O(n^{0.0000001})$.
- **Any polynomial is Big-Oh of any exponential.** $\forall a > 0, b > 1, n^a = O(b^n)$. For instance, $n^{1000} = O(1.0000001^n)$.

#### Example

Ranked in increasing order of growth:

1. $100n$. $100n = O(n \log_2 n)$. Let $c = 100, n_0 = 2$. $100n \leq 100n \log_2 n, \forall n \geq 2$. $1 \leq \log_2, \forall n \geq 2$.
2. $n \log_2 n$. $n \log_2 n = O(3^{\log_2 n} \approx n^{1.x}).n \log_2 n \leq c * n^{1.x}. \log_2 n \leq c * n^{0.x}$. Holds true since logarithms = O(exponentials).
3. $3^{\log_2 n}$. Applying $x^{\log_y z} = z^{\log_y x}$ rule: Since $x = y^{\log_y x}$, $3^{\log_2 n} = 2^{\log_2 3^{\log_2 n}} = 2^{\log_2 n * \log_2 3} = 2^{\log_2 n^{\log_2 3}} = n^{\log_2 3} \approx n^{1.x} = O(n^2)$.
4. $n^2$.

## 15.3 "Big-Omega" Notation: Best case

"Big-Omega" Notation

$f(n) = \Omega(g(n))$ if there exists $c \in (0, \infty)$ and $n_0 \in \mathbb{N}$ such that $f(n) \geq c * g(n)$ for every $n \geq n_0$.

- Asymptotic version of $f(n) \geq g(n)$.
- Roughly equivalent to $\lim_{n\to\infty} \frac{f(n)}{g(n)} > 0$.
- If $f(n) = O(g(n))$, then $g(n) = \Omega(f(n))$.
- When proving Big-Omega, generally let $c$ be small, 1 or 1/2, to deflate the $g(n)$.

## 15.4 "Big-Theta" Notation: Order of Growth

"Big-Theta" Notation

$f(n) = \Theta(g(n))$ if there exists $c_1 \leq c_2 \in (0, \infty)$ and $n_0 \in \mathbb{N}$ such that $c_2 * g(n) \geq f(n) \geq c_1 * g(n)$ for every $n \geq n_0$.

- Asymptotic version of $f(n) = g(n)$. There are infinite functions for $g(n)$. $f(n)$ has 1 growth rate, but we use $O/\Theta/\Omega$ to represent it.
- Roughly equivalent to $\lim_{n\to\infty} \frac{f(n)}{g(n)} \in (0, \infty)$.
- **Equivalent to** $f(n) = O(g(n))$ **AND** $g(n) = \Omega(g(n))$. Same upper bound and lower bound = have the same order of growth. Asymptotically they are equivalent. The limit converges to some value (neither 0 nor $\infty$)
- **More precise than** $O$ **or** $\Omega$, but $O$ is simpler.

(Big-Theta is the explicit order of growth. $g(n)$ has the EXACT same order of growth as $f(n)$, but there are an infinite number of functions that have the same order of growth.)

### 15.4.1 Proving a function is Big-Theta of another function

To prove Big-Theta, need Big-Oh and Big-Omega proofs.

- $30 \log_2 n + 45 = \Theta(\log_2 n)$.
  $\underline{O}$: Sum coefficients: $c = 75$. Choose log base for convenience: $n_0 = 2$. $30 \log_2 n + 45 \leq 75 \log_2 n, \forall n \geq 2$. $45 \leq 45 log_2 n, \forall n \geq 2$.
  $\underline{\Omega}$: Let $c = 20, n_0 = 1$. $30 \log_2 n + 45 \geq 20 \log_2 n, \forall n \geq 1$. $10 \log_2 n + 45 \geq 0, \forall n \geq 1$.
- $4n \log_2 2n = \Theta(n \log_2 n)$. $4n \log_2 2n = 4n(\log_2 2 + \log_2 n) = 4n + 4n \log_2 n$.
  $\underline{O}$: Let $c = 8, n_0 = 2$. $4n + 4n \log_2 n \leq 8n \log_2 n, \forall n \geq 2$. $4n \leq 4n \log_2 n$.
  $\underline{\Omega}$: Let $c = 2, n_0 = 2$. $4n + 4n \log_2 n \geq 2n \log_2 n, \forall n \geq 2$. $4n + 2n \log_2 n \geq 0$.
- $\sum_{i=1}^{n} i = \Theta(n^2)$. Arithmetic sum: $\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \frac{n^2}{2} + \frac{n}{2}$.
  $\underline{O}$: Let $c = 1, n_0 = 1$. $\frac{n^2}{2} + \frac{n}{/2} \leq n^2, \forall n >= 1$. $\frac{n}{2} \leq \frac{n^2}{2}$. $n \leq n^2$.
  $\underline{\Omega}$: Let $c = \frac{1}{2}, n_0 = 2$. $\frac{n^2}{2} + \frac{n}{2} \geq \frac{1}{2} n^2, \forall n \geq 2$. $\frac{n}{2} \geq 0$.

## 15.5 Prove or Disprove: if... then...

To prove, give an argument that works for all $f$ and $g$. To disprove, give a counterexample where antecedent is true but consequent is false.

1. If $f(n) = \Omega(g(n))$, then $2^{f(n)} = \Omega(2^{g(n)})$.
   False.
   *Counterexample.* Let $f(n) = n, g(n) = 2n$. $f(n) = \Omega(g(n))$ since they are the same function when $c = 1/2$. But $2^{f(n)} = 2^n, 2^{g(n)} = 2^{2n} = 4^n$, and $2^n \neq \Omega(4^n)$. For exponential functions, the base determines the rate of growth, so $4^n$ grows faster than $2^n$.

2. CHALLENGE: If $f(n) = O(g(n))$, then $\log_2(f(n)) = O(\log_2(g(n)))$.
   True. Show $f(n) \leq c * g(n), \forall n \geq n_0$, meaning that $\log_2(f(n)) \leq c' \log_2(g(n)), \forall n \geq n_0'$.
   {Receive points on the bolded statements, plus manipulating equations and finding a valid $c$ and $n_0$.}

   *Proof.* **Assume** $f(n) = O(g(n))$**. Then** $f(n) \leq c * g(n), \forall n \geq n_0$ **for some constants** $c, n_0$**.**
   $\log_2(f(n)) \leq \log_2(c * g(n)) = \log_2(c) + \log_2(g(n))$.
   **Show:** $\log_2(c) + \log_2(g(n)) \leq c' * \log_2(g(n)), \forall n \geq n_0'$**.** <u>We can pick $c'$ and $n_0'$, but not $c$ nor $n_0$.</u>

| | |
|---|---|
| $\log_2(c) + \log_2(g(n))$ | |
| $\leq \log_2(g(n)) * \log_2(c) + \log_2 g(n)$ | If $g(n) \geq 2$, then $\log_2(g(n)) \geq 1$. |
| | So we know $\log_2(g(n)) * \log_2(c) > \log_2(c)$. |
| $= \log_2(g(n)) * (\log_2(c) + 1)$ | Factoring $\log_2 g(n)$ out. |
| | $(\log_2(c) + 1)$ is a constant factor of $\log_2(g(n))$. Let $c' = \log_2(c) + 1$. |

   Let $c' = \log_2(c) + 1, n_0' = max(1, n0)$, where $n_0$ must be at least 1 so that $g(n) \geq 2$. □

## 15.6 EXTRA: "Little-Oh" Notation:

"Little" gets rid of '='.

**"Little-Oh" Notation**

$f(n) = o(g(n))$ if for every $c > 0$ there exists $n_0 \in \mathbb{N}$ such that $f(n) < c * g(n)$ for every $n \geq n_0$.

- Asymptotic version of $f(n) < g(n)$.
- Roughly equivalent to $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$.

## 15.7 EXTRA: "Little-Omega" Notation:

"Little" gets rid of '='.

**"Little-Omega" Notation**

$f(n) = \omega(g(n))$ if for every $c > 0$ there exists $n_0 \in \mathbb{N}$ such that $f(n) > c * g(n)$ for every $n \geq n_0$.

- Asymptotic version of $f(n) > g(n)$.
- Roughly equivalent to $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$.

# Chapter 16

# Recursion Trees

- Used to solve recurrences to get a closed form for the run-time. There is also a substitution method.
- Like a simulation of what is occurring in the recurrence.

**Steps:**

1. Draw levels 0-2. Determine piece size and # of pieces.
2. Find a pattern for level $i$, the generic formulas for piece size and # of pieces.
3. Find the bottom level by setting the generic piece size formula (level $i$) = base case, and solving for $i$.
4. Find work @ each level using non-recursive terms in $T(n)$. For level 0, work = the non-recursive terms in $T(n)$. For levels 1, 2, $i$, substitute the piece size in for $n$ and multiply by the number of pieces. The formula at level $i$ is the generic work per level.
5. Find the total work across all levels. $\sum_{i=\text{first level}}^{\text{last level}}$ work at level $i$. Evaluate the summation.
6. To find the runtime, find Big-Oh of the sum by dropping lower-order terms/constants.

**Note:** To prove work @ level, if you have levels 0, 1, 2 and show the pattern holds for those three levels, and you can say that the work formula is based on observing the pattern. Don't need to rigorously prove.

| Level | Piece size | Tree | # of pieces | Work @ level |
|---|---|---|---|---|
| 0 | | | | |
| 1 | | | | |
| 2 | | | | |
| i | | | | |
| last level | | | | |

## 16.1 Examples

- Mergesort
- Karatsuba's Algorithm
- The Master Theorem
- The Median Algorithm: Version 2

# Chapter 17

# The Master Theorem

Can sometimes be used instead of a recursion tree to solve recurrences. Works for recurrences of the form:

$$T(n) = \mathbf{a} * T(n/\mathbf{b}) + Cn^{\mathbf{d}}$$

- If $\left(\frac{a}{b^d}\right) > 1$: $T(n) = \Theta(n^{\log_b a})$.
- If $\left(\frac{a}{b^d}\right) = 1$: $T(n) = \Theta(n^d \log_b n)$.
- If $\left(\frac{a}{b^d}\right) < 1$: $T(n) = \Theta(n^d)$.
- **a:** Number of pieces to recurse on.
- **b:** Denominator of piece size to recurse on.
- **d:** Exponent of the running time for non-recursive operations ($O(n^d)$, work for a single recursive step).

Divide and Conquer algorithms tend to have this form.

## 17.1 Derivation

**Recursion Tree:**

| Level | Piece size | Tree | # of pieces | Work @ level |
|---|---|---|---|---|
| 0 | $n$ | $\boxed{n}$ | 1 | $cn^d$ |
| 1 | $\frac{n}{b}$ | $\boxed{\frac{n}{b}} \dots$ | $a$ | $ac(\frac{n}{b})^d$ |
| 2 | $\frac{n}{b^2}$ | | $a^2$ | $a^2c(\frac{n}{b^2})^d$ |
| i | $\frac{n}{b^i}$ | | $a^i$ | $a^ic(\frac{cn}{b^i})^d$ |
| ... | | | | |
| $\log_b n$ (last level) | | | | |

Finding the last level: Assume $T(1) = c$.

$$\begin{aligned} \frac{n}{b^i} &= 1 \\ n &= b^i \\ \log_b n &= i && \text{Last level.} \end{aligned}$$

**Total work across all levels:** $\sum_{i=0}^{\log_b n} a^i c(\frac{n}{b^i})^d = cn^d \sum_{i=0}^{\log_b n} (\frac{a}{b^d})^i$. This is a geometric series. Therefore:
$S = \Theta(1)$ when $r < 1$.
$S = \Theta(r^l)$ when $r > 1$.
**Running time:**

- If $\left(\frac{a}{b^d}\right) > 1$, $cn^d \left(\frac{a}{b^d}\right)^{\log_b n} = cn^d \frac{a^{\log_b n}}{n^d} = cn^{\log_b a} = O(n^{\log_b a})$.
- If $\left(\frac{a}{b^d}\right) = 1$, every term in the sum is 2. There are $\log_b n+1$ terms, so $cn^d(\log_b n+1) = cn^2 \log_b n + cn^d = O(n^d \log_b n)$.
- If $\left(\frac{a}{b^d}\right) < 1$, the sum behaves like a constant, $c'$. Then $cn^d c' = O(n^d)$.

## 17.2 EXTRA: Even More General

Works for recurrences of the form:

$$T(n) = \mathbf{a} * T(n/\mathbf{b}) + f(n)$$

- If $f(n) = O(n^{(\log_b a)-\mathcal{E}})$: $T(n) = \Theta(n^{\log_b a})$.
- If $f(n) = O(n^{\log_b a})$: $T(n) = \Theta(f(n) * \log n)$.
- If $f(n) = \Omega(n^{(\log_b a)+\mathcal{E}}) \wedge af(\frac{n}{b}) \leq Cf(n)$ for $C < 1$: $T(n) = \Theta(f(n))$.

## 17.3 Examples

- $T(n) = 16 * T(\frac{n}{4}) + n^2$. $\frac{a}{b^d} = \frac{16}{4^2} = 1$. Then $\Theta(n^2 \log_4 n)$.
- $T(n) = 21 * T(\frac{n}{5}) + n^2$. $\frac{a}{b^d} = \frac{21}{5^2} = 1$. Then $\Theta(n^2)$.
- $T(n) = 2 * T(\frac{n}{2}) + 1$. $\frac{a}{b^d} = \frac{2}{2^0} > 1$. Then $\Theta(n^{\log_2 2} = n)$.
- $T(n) = 1 * T(\frac{n}{2}) + 1$. $\frac{a}{b^d} = \frac{1}{2^0} = 1$. Then $\Theta(n^0 \log_2 n = \log_2 n)$.

### 17.3.1 Mergesort

**Recurrence:** $T(n) = 2 * T(n/2) + Cn = O(n \log n)$

- **a:** 2. Recurses on 2 pieces.
- **b:** 2. Recursed on pieces of size $n/2$.
- **d:** 1. Merge took linear time.

### 17.3.2 Karatsuba

**Recurrence:** $T(n) = 3T(\frac{n}{2}) + cn = O(n^{\log_2 3}$

- **a:** 3. Recurses on 3 pieces.
- **b:** 2. Recursed on pieces of size $n/2$.
- **d:** 1. Combination took linear time.

# Chapter 18

# Equations

### 18.0.1 Logarithms

$\log_a b = \dfrac{\log_x b}{log_x a}$ Change of base:

$x^{\log_y z} = z^{\log_y x}$ since $x^{\log_y z} = y^{\log_y x^{\log_y z}} = y^{log_y z * \log_y x} = y^{log_y z^{\log_y x}} = z^{\log_y x}$

### 18.0.2 Reccurences

Geometric Series ($r \neq 1, r > 0$): $S = \sum_{i=0}^{l} r^l$. Be able to identify.
$S = \Theta(1)$ when $r < 1$.
$S = \Theta(r^l)$ when $r > 1$.
EXTRA Solution: $S = \frac{1-r^{l+1}}{1-r} = \frac{r^{l+1}-1}{r-1}$
EXTRA Derivation:

$$\begin{aligned}
S &= 1 + r + r^2 + ... + r^l \\
rS &= \quad\ r + r^2 + ... + r^l + r^{l+1} \\
S(1-r) &= S - rS = 1 - r^{l+1} \\
S(r-1) &= rS - S = r^{l+1} - 1
\end{aligned}$$

## 18.1 TO ADD:

Arithmetic series sum.

## 18.2 Missing parts:

- Recitation 2: Recursion tree, Divide and Conquer Algorithm, Master Theorem
- Lecture 11: DP practice: Sauron.