Macacário Maratona de Programação v3.3
Instituto Tecnológico de Aeronáutica

Lucas França de Oliveira (Mexicano T-18)
lucas.fra.oli18@gmail.com

22 de fevereiro de 2024

Sumário

Capítulo 1

Introdução

1.1 Bugs do Milênio

Erros teóricos:

- Não ler o enunciado do problema com calma.
- Assumir algum fato sobre a solução na pressa.
- Não reler os limites do problema antes de submeter.
- Quando adaptar um algoritmo, atentar para todos os detalhes da estrutura do algoritmo, se devem (ou não) ser modificados (ex:marcação de vértices/estados).
- O problema pode ser NP, disfarçado ou mesmo sem limites especificados. Nesse caso a solução é bronca mesmo. Não é hora de tentar ganhar o prêmio nobel.

Erros com valor máximo de variável:

- Verificar com calma (fazer as contas direito) para ver se o infinito é tão infinito quanto parece.
- Verificar se operações com infinito estouram 31 bits.
- Usar multiplicacao de *int*'s e estourar 32 bits (por exemplo, checar sinais usando $a * b > 0$).

Erros de casos extremos:

- Testou caso $n = 0$? $n = 1$? $n = MAXN$? Muitas vezes tem que tratar separado.
- Pense em todos os casos que podem ser considerados casos extremos ou casos isolados.
- Casos extremos podem atrapalhar não só no algoritmo, mas em coisas como construir alguma estrutura (ex: lista de adj em grafos).
- Não esquecer de self-loops ou multiarestas em grafos.
- Em problemas de caminho Euleriano, verificar se o grafo é conexo.

Erros de desatenção em implementação:

- Errar ctrl-C/ctrl-V em código. Muito comum.
- Colocar igualdade dentro de if? ($if(a = 0)continue$;)
- Esquecer de inicializar variável.
- Trocar *break* por *continue* (ou vice-versa).
- Declarar variável global e variável local com mesmo nome (é pedir pra dar merda...).

Erros de implementação:

- Definir variável com tipo errado (*int* por *double*, *int* por *char*).
- Não usar variável com nome max e min.
- Não esquecer que $.size()$ é unsigned.
- Lembrar que 1 é *int*, ou seja, se fizer *long long* $a = 1 << 40$;, não irá funcionar (o ideal é fazer *long long* $a = 1LL << 40$;).

Erros em limites:

- Qual o ordem do tempo e memória? 10^8 é uma referência para tempo. Sempre verificar rapidamente a memória, apesar de que o limite costuma ser bem grande.
- A constante pode ser muito diminuída com um algoritmo melhor (ex: húngaro no lugar de fluxo) ou com operações mais rápidas (ex: divisões são lentas, bitwise é rápido)?
- O exercício é um caso particular que pode (e está precisando) ser otimizado e não usar direto a biblioteca?

Erros em doubles:

- Primeiro, evitar (a não ser que seja necessário ou mais simples a solução) usar $float/double$. E.g. conta que só precisa de 2 casas decimais pode ser feita com inteiro e depois %100.
- Sempre usar *double*, não $float$ (a não ser que o enunciado peça explicitamente).
- Testar igualdade com tolerância (absoluta, e talvez relativa).
- Cuidado com erros de imprecisão, em particular evitar ao máximo subtrair dois números praticamente iguais.

Outros erros:

- Evitar (a não ser que seja necessário) alocação dinâmica de memória.

- Não usar STL desnecessariamente (ex: vector quando um array normal dá na mesma), mas usar se facilitar (ex: nomes associados a vértices de um grafo - $map < string, int >$) ou se precisar (ex: um algoritmo $O(n \log n)$ que usa $< set >$ é necessário para passar no tempo).
- Não inicializar variável a cada teste (zerou vetores? zerou variável que soma algo? zerou com zero? era pra zerar com zero, com -1 ou com INF?).
- Saída está formatada corretamente?
- Declarou vetor com tamanho suficiente?
- Cuidado ao tirar o módulo de número negativo. Ex.: $x\%n$ não dá o resultado esperado se x é negativo, fazer $(x\%n + n)\%n$.

1.2 Os 1010 mandamentos

Cortesia da PUC-RJ.

0. Não dividirás por zero.
1. Não alocarás dinamicamente.
2. Compararás números de ponto flutuante usando EPS.
3. Verificarás se o grafo pode ser desconexo.
4. Verificarás se as arestas do grafo podem ter peso negativo.
5. Verificarás se pode haver mais de uma aresta ligando dois vértices.
6. Conferirás todos os índices de uma programação dinâmica.
7. Reduzirás o branching factor da DFS.
8. Farás todos os cortes possíveis em uma DFS.
9. Tomarás cuidado com pontos coincidentes e com pontos colineares.

1.3 Truques sujos (porém válidos)

- **Método Steve Halim**: As possíveis saídas do problema cabem no código do problema? Deixe um algoritmo *naive* brutando o problema na máquina por alguns minutos e escreva as respostas direto no código para submeter. Exemplo: problema cuja entrada é um único número da ordem de 10^5. Verificar o tamanho máximo de caracteres de uma submissão.
- **Fatoriais até** 10^9: Deixe um programa na sua máquina brutando os fatoriais até 10^9. A cada 10^3 ou 10^6, imprima. Cole a saída no código e use os valores pré-calculados pra calcular um fatorial com 10^3 ou 10^6 operações.
- **Problemas com constantes**: Se algum valor útil de algum problema for constante (independe da entrada), mas você não sabe, brute ele na sua máquina e cole no código.
- **Debug com assert**: Pode colocar *assert* em código para submeter. Tente usar isso pra transformar um WA em um RTE. É uma forma válida de debug. Usar isso somente no desespero (fica gastando submissões).

1.4 Limites da representação de dados

tipo	scanf	bits	mínimo	..	máximo	precisão decimal
char	$\%c$	8	0	..	255	2
signed char	$\%hhd$	8	-128	..	127	2
unsigned char	$\%hhu$	8	0	..	255	2
short	$\%hd$	16	-32.768	..	32.767	4
unsigned short	$\%hu$	16	0	..	65.535	4
int	$\%d$	32	-2×10^9	..	2×10^9	9
unsigned int	$\%u$	32	0	..	4×10^9	9
long long	$\%lld$	64	-9×10^{18}	..	9×10^{18}	18
unsigned long long	$\%llu$	64	0	..	18×10^{18}	19

tipo	scanf	bits	expoente	precisão decimal
float	$\%f$	32	38	6
double	$\%lf$	64	308	15
long double	$\%Lf$	80	19.728	18

1.5 Quantidade de números primos de 1 até 10^n

É sempre verdade que $n/ln(n) < pi(n) < 1.26 * n/ln(n)$.

$pi(10^1) = 4$	$pi(10^2) = 25$	$pi(10^3) = 168$
$pi(10^4) = 1.229$	$pi(10^5) = 9.592$	$pi(10^6) = 78.498$
$pi(10^7) = 664.579$	$pi(10^8) = 5.761.455$	$pi(10^9) = 50.847.534$

1.6 Triângulo de Pascal

$n\backslash p$	0	1	2	3	4	5	6	7	8	9	10
0	1										
1	1	1									
2	1	2	1								
3	1	3	3	1							
4	1	4	6	4	1						
5	1	5	10	10	5	1					
6	1	6	15	20	15	6	1				
7	1	7	21	35	35	21	7	1			
8	1	8	28	56	70	56	28	8	1		
9	1	9	36	84	126	126	84	36	9	1	
10	1	10	45	120	210	252	210	120	45	10	1

$C(33, 16)$	1.166.803.110	limite do int
$C(34, 17)$	2.333.606.220	limite do unsigned int
$C(66, 33)$	7.219.428.434.016.265.740	limite do long long
$C(67, 33)$	14.226.520.737.620.288.370	limite do unsigned long long

1.7 Fatoriais

Fatoriais até 20 com os limites de tipo.

0!	1	
1!	1	
2!	2	
3!	6	
4!	24	
5!	120	
6!	720	
7!	5.040	
8!	40.320	
9!	362.880	
10!	3.628.800	
11!	39.916.800	
12!	479.001.600	limite do unsigned int
13!	6.227.020.800	
14!	87.178.291.200	
15!	1.307.674.368.000	
16!	20.922.789.888.000	
17!	355.687.428.096.000	
18!	6.402.373.705.728.000	
19!	121.645.100.408.832.000	
20!	2.432.902.008.176.640.000	limite do unsigned long long

1.8 Tabela ASCII

Char	Dec	Oct	Hex	Char	Dec	Oct	Hex	Char	Dec	Oct	Hex	Char	Dec	Oct	Hex
(nul)	0	0000	0x00	(sp)	32	0040	0x20	@	64	0100	0x40	`	96	0140	0x60
(soh)	1	0001	0x01	!	33	0041	0x21	**A**	**65**	**0101**	**0x41**	**a**	**97**	**0141**	**0x61**
(stx)	2	0002	0x02	"	34	0042	0x22	**B**	**66**	**0102**	**0x42**	**b**	**98**	**0142**	**0x62**
(etx)	3	0003	0x03	#	35	0043	0x23	**C**	**67**	**0103**	**0x43**	**c**	**99**	**0143**	**0x63**
(eot)	4	0004	0x04	$	36	0044	0x24	**D**	**68**	**0104**	**0x44**	**d**	**100**	**0144**	**0x64**
(enq)	5	0005	0x05	%	37	0045	0x25	**E**	**69**	**0105**	**0x45**	**e**	**101**	**0145**	**0x65**
(ack)	6	0006	0x06	&	38	0046	0x26	**F**	**70**	**0106**	**0x46**	**f**	**102**	**0146**	**0x66**
(bel)	7	0007	0x07	'	39	0047	0x27	**G**	**71**	**0107**	**0x47**	**g**	**103**	**0147**	**0x67**
(bs)	8	0010	0x08	(	40	0050	0x28	**H**	**72**	**0110**	**0x48**	**h**	**104**	**0150**	**0x68**
(ht)	9	0011	0x09	)	41	0051	0x29	**I**	**73**	**0111**	**0x49**	**i**	**105**	**0151**	**0x69**
(nl)	10	0012	0x0a	*	42	0052	0x2a	**J**	**74**	**0112**	**0x4a**	**j**	**106**	**0152**	**0x6a**
(vt)	11	0013	0x0b	+	43	0053	0x2b	**K**	**75**	**0113**	**0x4b**	**k**	**107**	**0153**	**0x6b**
(np)	12	0014	0x0c	,	44	0054	0x2c	**L**	**76**	**0114**	**0x4c**	**l**	**108**	**0154**	**0x6c**
(cr)	13	0015	0x0d	-	45	0055	0x2d	**M**	**77**	**0115**	**0x4d**	**m**	**109**	**0155**	**0x6d**
(so)	14	0016	0x0e	.	46	0056	0x2e	**N**	**78**	**0116**	**0x4e**	**n**	**110**	**0156**	**0x6e**
(si)	15	0017	0x0f	/	47	0057	0x2f	**O**	**79**	**0117**	**0x4f**	**o**	**111**	**0157**	**0x6f**
(dle)	16	0020	0x10	**0**	**48**	**0060**	**0x30**	**P**	**80**	**0120**	**0x50**	**p**	**112**	**0160**	**0x70**
(dc1)	17	0021	0x11	**1**	**49**	**0061**	**0x31**	**Q**	**81**	**0121**	**0x51**	**q**	**113**	**0161**	**0x71**
(dc2)	18	0022	0x12	**2**	**50**	**0062**	**0x32**	**R**	**82**	**0122**	**0x52**	**r**	**114**	**0162**	**0x72**
(dc3)	19	0023	0x13	**3**	**51**	**0063**	**0x33**	**S**	**83**	**0123**	**0x53**	**s**	**115**	**0163**	**0x73**
(dc4)	20	0024	0x14	**4**	**52**	**0064**	**0x34**	**T**	**84**	**0124**	**0x54**	**t**	**116**	**0164**	**0x74**
(nak)	21	0025	0x15	**5**	**53**	**0065**	**0x35**	**U**	**85**	**0125**	**0x55**	**u**	**117**	**0165**	**0x75**
(syn)	22	0026	0x16	**6**	**54**	**0066**	**0x36**	**V**	**86**	**0126**	**0x56**	**v**	**118**	**0166**	**0x76**
(etb)	23	0027	0x17	**7**	**55**	**0067**	**0x37**	**W**	**87**	**0127**	**0x57**	**w**	**119**	**0167**	**0x77**
(can)	24	0030	0x18	**8**	**56**	**0070**	**0x38**	**X**	**88**	**0130**	**0x58**	**x**	**120**	**0170**	**0x78**
(em)	25	0031	0x19	**9**	**57**	**0071**	**0x39**	**Y**	**89**	**0131**	**0x59**	**y**	**121**	**0171**	**0x79**
(sub)	26	0032	0x1a	:	58	0072	0x3a	**Z**	**90**	**0132**	**0x5a**	**z**	**122**	**0172**	**0x7a**
(esc)	27	0033	0x1b	;	59	0073	0x3b	[	91	0133	0x5b	{	123	0173	0x7b
(fs)	28	0034	0x1c	<	60	0074	0x3c	\	92	0134	0x5c	|	124	0174	0x7c
(gs)	29	0035	0x1d	=	61	0075	0x3d	]	93	0135	0x5d	}	125	0175	0x7d
(rs)	30	0036	0x1e	>	62	0076	0x3e	^	94	0136	0x5e	~	126	0176	0x7e
(us)	31	0037	0x1f	?	63	0077	0x3f	_	95	0137	0x5f	(del)	127	0177	0x7f

1.9 Primos até 10.000

Existem 1.229 números primos até 10.000.

2	3	5	7	11	13	17	19	23	29	31
37	41	43	47	53	59	61	67	71	73	79
83	89	97	101	103	107	109	113	127	131	137
139	149	151	157	163	167	173	179	181	191	193
197	199	211	223	227	229	233	239	241	251	257
263	269	271	277	281	283	293	307	311	313	317
331	337	347	349	353	359	367	373	379	383	389
397	401	409	419	421	431	433	439	443	449	457
461	463	467	479	487	491	499	503	509	521	523
541	547	557	563	569	571	577	587	593	599	601
607	613	617	619	631	641	643	647	653	659	661
673	677	683	691	701	709	719	727	733	739	743
751	757	761	769	773	787	797	809	811	821	823
827	829	839	853	857	859	863	877	881	883	887
907	911	919	929	937	941	947	953	967	971	977
983	991	997	1009	1013	1019	1021	1031	1033	1039	1049
1051	1061	1063	1069	1087	1091	1093	1097	1103	1109	1117
1123	1129	1151	1153	1163	1171	1181	1187	1193	1201	1213
1217	1223	1229	1231	1237	1249	1259	1277	1279	1283	1289
1291	1297	1301	1303	1307	1319	1321	1327	1361	1367	1373
1381	1399	1409	1423	1427	1429	1433	1439	1447	1451	1453
1459	1471	1481	1483	1487	1489	1493	1499	1511	1523	1531
1543	1549	1553	1559	1567	1571	1579	1583	1597	1601	1607
1609	1613	1619	1621	1627	1637	1657	1663	1667	1669	1693
1697	1699	1709	1721	1723	1733	1741	1747	1753	1759	1777
1783	1787	1789	1801	1811	1823	1831	1847	1861	1867	1871
1873	1877	1879	1889	1901	1907	1913	1931	1933	1949	1951
1973	1979	1987	1993	1997	1999	2003	2011	2017	2027	2029
2039	2053	2063	2069	2081	2083	2087	2089	2099	2111	2113
2129	2131	2137	2141	2143	2153	2161	2179	2203	2207	2213
2221	2237	2239	2243	2251	2267	2269	2273	2281	2287	2293
2297	2309	2311	2333	2339	2341	2347	2351	2357	2371	2377
2381	2383	2389	2393	2399	2411	2417	2423	2437	2441	2447
2459	2467	2473	2477	2503	2521	2531	2539	2543	2549	2551
2557	2579	2591	2593	2609	2617	2621	2633	2647	2657	2659
2663	2671	2677	2683	2687	2689	2693	2699	2707	2711	2713
2719	2729	2731	2741	2749	2753	2767	2777	2789	2791	2797
2801	2803	2819	2833	2837	2843	2851	2857	2861	2879	2887
2897	2903	2909	2917	2927	2939	2953	2957	2963	2969	2971
2999	3001	3011	3019	3023	3037	3041	3049	3061	3067	3079
3083	3089	3109	3119	3121	3137	3163	3167	3169	3181	3187
3191	3203	3209	3217	3221	3229	3251	3253	3257	3259	3271
3299	3301	3307	3313	3319	3323	3329	3331	3343	3347	3359
3361	3371	3373	3389	3391	3407	3413	3433	3449	3457	3461
3463	3467	3469	3491	3499	3511	3517	3527	3529	3533	3539
3541	3547	3557	3559	3571	3581	3583	3593	3607	3613	3617
3623	3631	3637	3643	3659	3671	3673	3677	3691	3697	3701
3709	3719	3727	3733	3739	3761	3767	3769	3779	3793	3797
3803	3821	3823	3833	3847	3851	3853	3863	3877	3881	3889
3907	3911	3917	3919	3923	3929	3931	3943	3947	3967	3989
4001	4003	4007	4013	4019	4021	4027	4049	4051	4057	4073
4079	4091	4093	4099	4111	4127	4129	4133	4139	4153	4157

4159	4177	4201	4211	4217	4219	4229	4231	4241	4243	4253
4259	4261	4271	4273	4283	4289	4297	4327	4337	4339	4349
4357	4363	4373	4391	4397	4409	4421	4423	4441	4447	4451
4457	4463	4481	4483	4493	4507	4513	4517	4519	4523	4547
4549	4561	4567	4583	4591	4597	4603	4621	4637	4639	4643
4649	4651	4657	4663	4673	4679	4691	4703	4721	4723	4729
4733	4751	4759	4783	4787	4789	4793	4799	4801	4813	4817
4831	4861	4871	4877	4889	4903	4909	4919	4931	4933	4937
4943	4951	4957	4967	4969	4973	4987	4993	4999	5003	5009
5011	5021	5023	5039	5051	5059	5077	5081	5087	5099	5101
5107	5113	5119	5147	5153	5167	5171	5179	5189	5197	5209
5227	5231	5233	5237	5261	5273	5279	5281	5297	5303	5309
5323	5333	5347	5351	5381	5387	5393	5399	5407	5413	5417
5419	5431	5437	5441	5443	5449	5471	5477	5479	5483	5501
5503	5507	5519	5521	5527	5531	5557	5563	5569	5573	5581
5591	5623	5639	5641	5647	5651	5653	5657	5659	5669	5683
5689	5693	5701	5711	5717	5737	5741	5743	5749	5779	5783
5791	5801	5807	5813	5821	5827	5839	5843	5849	5851	5857
5861	5867	5869	5879	5881	5897	5903	5923	5927	5939	5953
5981	5987	6007	6011	6029	6037	6043	6047	6053	6067	6073
6079	6089	6091	6101	6113	6121	6131	6133	6143	6151	6163
6173	6197	6199	6203	6211	6217	6221	6229	6247	6257	6263
6269	6271	6277	6287	6299	6301	6311	6317	6323	6329	6337
6343	6353	6359	6361	6367	6373	6379	6389	6397	6421	6427
6449	6451	6469	6473	6481	6491	6521	6529	6547	6551	6553
6563	6569	6571	6577	6581	6599	6607	6619	6637	6653	6659
6661	6673	6679	6689	6691	6701	6703	6709	6719	6733	6737
6761	6763	6779	6781	6791	6793	6803	6823	6827	6829	6833
6841	6857	6863	6869	6871	6883	6899	6907	6911	6917	6947
6949	6959	6961	6967	6971	6977	6983	6991	6997	7001	7013
7019	7027	7039	7043	7057	7069	7079	7103	7109	7121	7127
7129	7151	7159	7177	7187	7193	7207	7211	7213	7219	7229
7237	7243	7247	7253	7283	7297	7307	7309	7321	7331	7333
7349	7351	7369	7393	7411	7417	7433	7451	7457	7459	7477
7481	7487	7489	7499	7507	7517	7523	7529	7537	7541	7547
7549	7559	7561	7573	7577	7583	7589	7591	7603	7607	7621
7639	7643	7649	7669	7673	7681	7687	7691	7699	7703	7717
7723	7727	7741	7753	7757	7759	7789	7793	7817	7823	7829
7841	7853	7867	7873	7877	7879	7883	7901	7907	7919	7927
7933	7937	7949	7951	7963	7993	8009	8011	8017	8039	8053
8059	8069	8081	8087	8089	8093	8101	8111	8117	8123	8147
8161	8167	8171	8179	8191	8209	8219	8221	8231	8233	8237
8243	8263	8269	8273	8287	8291	8293	8297	8311	8317	8329
8353	8363	8369	8377	8387	8389	8419	8423	8429	8431	8443
8447	8461	8467	8501	8513	8521	8527	8537	8539	8543	8563
8573	8581	8597	8599	8609	8623	8627	8629	8641	8647	8663
8669	8677	8681	8689	8693	8699	8707	8713	8719	8731	8737
8741	8747	8753	8761	8779	8783	8803	8807	8819	8821	8831
8837	8839	8849	8861	8863	8867	8887	8893	8923	8929	8933
8941	8951	8963	8969	8971	8999	9001	9007	9011	9013	9029
9041	9043	9049	9059	9067	9091	9103	9109	9127	9133	9137
9151	9157	9161	9173	9181	9187	9199	9203	9209	9221	9227
9239	9241	9257	9277	9281	9283	9293	9311	9319	9323	9337
9341	9343	9349	9371	9377	9391	9397	9403	9413	9419	9421
9431	9433	9437	9439	9461	9463	9467	9473	9479	9491	9497
9511	9521	9533	9539	9547	9551	9587	9601	9613	9619	9623
9629	9631	9643	9649	9661	9677	9679	9689	9697	9719	9721
9733	9739	9743	9749	9767	9769	9781	9787	9791	9803	9811
9817	9829	9833	9839	9851	9857	9859	9871	9883	9887	9901
9907	9923	9929	9931	9941	9949	9967	9973			

Capítulo 2

C++ e Biblioteca STD

2.1 Macros

```
#include <bits/stdc++.h>
#define DEBUG false
#define debugf if (DEBUG) printf
#define MAXN 200309
#define MAXM 900009
#define ALFA 256
#define MOD 1000000007
#define INF 0x3f3f3f3f
#define INFLL 0x3f3f3f3f3f3f3f3f
#define EPS 1e-9
#define PI 3.14159265358979323846264338327950288 4
#define FOR(x,n) for(int x=0; (x)<int(n); (x)++)
#define FOR1(x,n) for(int x=1; (x)<=int(n); (x)++)
#define REP(x,n) for(int x=int(n)-1; (x)>=0; (x)--)
#define REP1(x,n) for(int x=(n); (x)>0; (x)--)
#define pb push_back
#define pf push_front
#define fi first
#define se second
#define mp make_pair
#define sz(x) int(x.size())
#define all(x) x.begin(), x.end()
#define mset(x,y) memset(&x, (y), sizeof(x))
using namespace std;
typedef long long ll;
typedef unsigned long long ull;
typedef long double ld;
typedef unsigned int uint;
typedef vector<int> vi;
typedef pair<int, int> ii;
```

2.2 Compilador GNU

Alguns comandos do compilador do GNU traduz para algumas instruções em Assembly (muito rápido).

__builtin_ffs(int) //Retorna 1 + posição do bit 1 menos significativo. Retorna zero para 0.
__builtin_clz(int) //Retorna o número de zeros na frente do bit 1 mais significativo. Não definido para zero.
__builtin_ctz(int) //Retorna o número de zeros atrás do bit 1 menos significativo. Não definido para zero.
__builtin_popcount(int) //Soma dos bits.
__builtin_parity(int) //Soma dos bits módulo 2.

__builtin_ffsll(long long) //Retorna 1 + posição do bit 1 menos significativo. Retorna zero para 0.
__builtin_clzll(long long) //Retorna o número de zeros na frente do bit 1 mais significativo. Não definido para zero.
__builtin_ctzll(long long) //Retorna o número de zeros atrás do bit 1 menos significativo. Não definido para zero.
__builtin_popcountll(long long) //Soma dos bits.
__builtin_parityll(long long) //Soma dos bits módulo 2.

2.3 I/O rápido

Para acelear *cin/cout*: **ios_base::sync_with_stdio(0)**

gets é mais rápido que *scanf*, que é mais rápido que *cin*. JAMAIS usar *gets* fora de programação competitiva. *gets* não lê '\n', mas para quando o encontra. *puts* é mais rápido que *printf*, que é mais rápido que *cout*. *puts* coloca o '\n' ao final.

Para preprocessar todo o input rápido:

```
#define MAXI 100000009
char input[MAXI];
void readInput() {
    input[fread(input, 1, sizeof(input)-4, stdin)] =
        0;
}
```

Para ler inteiros rápido (não serve para programas que acabam em EOF):

```
int readInt() {
    bool minus = false;
    int ans = 0;
    char c;
    while (c = getchar()) {
        if (c == '-') break;
        if (c >= '0' && c <= '9') break;
    }
    if (c == '-') minus = true;
    else ans = c-'0';
    while (c = getchar()) {
        if (c < '0' || c > '9') break;
        ans = ans*10 + (c - '0');
    }
    return minus ? -ans : ans;
}
```

2.4 Verificar overflow

```
if (b > 0 && a > INFLL-b) //a+b vai dar overflow
if (b < 0 && a < -INFLL-b) //a+b vai dar underflow
if (b < 0 && a > INFLL+b) //a-b vai dar overflow
if (b > 0 && a < -INFLL+b) //a-b vai dar underflow
if (b > INFLL/a) //a*b vai dar overflow
if (b < -INFLL/a) //a*b vai dar underflow
```

2.5 C++11

auto a = b //a é o tipo de b.
auto a = b() //a é o tipo de retorno de b.
for(T a : b) //itera sobre todos os elementos de uma coleção iterável b.
for(T & a : b) //itera sobre todas as referências de uma coleção iterável b.
lambda functions: [] (params) -> type {body} retorna o ponteiro para uma função **type name(params) {body}**

2.6 Complex

Exemplo: #**include** <**complex**>, **complex**<**double**> **point**;
Funções: **real**, **imag**, **abs**, **arg**, **norm**, **conj**, **polar**

2.7 Pair

#**include** <**utility**>
pair<**tipo1, tipo2**> **P;**
tipo1 first, tipo2 second

2.8 List

list<**Elem**> **c** //Cria uma lista vazia.
list<**Elem**> **c1(c2)** //Cria uma cópia de uma outra lista do mesmo tipo (todos os elementos são copiados).
list<**Elem**> **c(n)** //Cria uma lista com n elementos definidos pelo construtor default.
list<**Elem**> **c(n,elem)** //Cria uma lista inicializada com n cópias do elemento $elem$.
list<**Elem**> **c(beg,end)** //Cria uma lista com os elementos no intervalo $[beg, end)$.
c.~list<**Elem**>**()** //Destrói todos os elementos e libera a memória.

Membros de list:
begin, **end**, **rbegin**, **rend**, **size**, **empty**, **clear**, **swap**.
front //Retorna o primeiro elemento.
back //Retorna o último elemento.
push_back //Coloca uma cópia de $elem$ no final da lista.
pop_back //Remove o último elemento e não retorna ele.
push_front //Insere uma cópia de $elem$ no começo da lista.
pop_front //Remove o primeiro elemento da lista e não retorna ele.
swap //Troca duas list's em $O(1)$.
erase (it)//Remove o elemento na posição apontada pelo iterador it e retorna a posição do próximo elemento.
erase (beg,end)//Remove todos os elementos no range $[beg, end)$ e retorna a posição do próximo elemento;
insert (it, pos)//Insere o elemento pos na posição anterior à apontada pelo iterador it.

2.9 Vector

#**include** <**vector**>
vector<**tipo**> **V;**

Membros de vector:
begin, **end**, **rbegin**, **rend**, **size**, **empty**, **clear**, **swap**.
reserve //Seta a capacidade mínima do vetor.
front //Retorna a referência para o primeiro elemento.
back //Retorna a referência para o último elemento.
erase //Remove um elemento do vetor.
pop_back //Remove o último elemento do vetor.
push_back //Adiciona um elemento no final do vetor.
swap //Troca dois vector's em $O(1)$.

2.10 Deque

#**include** <**queue**>
deque<**tipo**> **Q;**
Q[50] //Acesso randômico.

Membros de deque:
begin, **end**, **rbegin**, **rend**, **size**, **empty**, **clear**, **swap**.
front //Retorna uma referência para o primeiro elemento.
back //retorna uma referência para o último elemento.
erase //Remove um elemento do deque.
pop_back //Remove o último elemento do deque.
pop_front //Remove o primeiro elemento do deque.
push_back //Insere um elemento no final do deque.
push_front//Insere um elemento no começo do deque.

2.11 Queue

#**include** <**queue**>
queue<**tipo**> **Q;**

Membros de queue:
back //Retorna uma referência ao último elemento da fila.
empty //Retorna se a fila está vazia ou não.
front //Retorna uma referência ao primeiro elemento da fila.
pop //Retorna o primeiro elemento da fila.
push //Insere um elemento no final da fila.
size //Retorna o número de elementos da fila.

2.12 Stack

#**include** <**stack**>
stack<**tipo**> **P;**

Membros de stack:
empty //Retorna se pilha está vazia ou não.
pop //Remove o elemento no topo da pilha.
push //Insere um elemento na pilha.
size //retorna o tamanho da pilha.
top //Retorna uma referência para o elemento no topo da pilha.

2.13 Map

#include <map>
#include <string>
map<string, int> si;

Membros de map:
begin, **end**, **rbegin**, **rend**, **size**, **empty**, **clear**, **swap**, **count**.
erase //Remove um elemento do mapa.
find //retorna um iterador para um elemento do mapa que tenha a chave.
lower_bound //Retorna um iterador para o primeiro elemento maior que a chave ou igual à chave.
upper_bound //Retorna um iterador para o primeiro elemento maior que a chave.

Map é um set de pair, ao iterar pelos elementos de map, $i->first$ é a chave e $i->second$ é o valor.
Map com comparador personalizado: Utilizar **struct** com **bool operator<(tipoStruct s) const** . Cuidado pra diferenciar os elementos!

2.14 Set

#include <set>
set<tipo> S;

Membros de set:
begin, **end**, **rbegin**, **rend**, **size**, **empty**, **clear**, **swap**.
erase //Remove um elemento do set.
find //Retorna um iterador para um elemento do set.
insert //Insere um elemento no set.
lower_bound //Retorna um iterador para o primeiro elemento maior que um valor ou igual a um valor.
upper_bound //Retorna um iterador para o primeiro elemento maior que um valor.

Criando set com comparador personalizado: Utilizar **struct cmp** com **bool operator()(tipo, tipo) const** e declarar **set<tipo, vector<tipo>, cmp()> S**. Cuidado pra diferenciar os elementos!

2.15 Ordered set

```
#include <ext/pb_ds/tree_policy.hpp>
#include <ext/pb_ds/assoc_container.hpp>
using namespace __gnu_pbds;
typedef tree<int,null_type,less<int>,rb_tree_tag,
tree_order_statistics_node_update> ordered_set;
```

Membros de ordered_set:
find_by_order(p) //Retorna um ponteiro para o p-ésimo elemento do set. Se p é maior que o tamanho de n, retorna o fim do set.
order_by_key(v) //Retorna quantos elementos são menores que v.

Mesmo set com operações de find_by_order e order_by_key.

2.16 Unordered set e map

Igual a set e map, porém usa Hash Table (é mais rápido). Precisa de C++11.
unordered_set<tipo> S;
unordered_map<chave, valor> S;

2.17 Priority Queue

#include <queue>
priority_queue<tipo> pq

Membros: **empty**, **size**, **top**, **push**, **pop**.
Utilizar **struct cmp** com **bool operator ()(tipo, tipo)** e declarar **priority_queue<tipo, vector<tipo>, cmp()> pq**
.

Maior vem antes!

2.18 Bitset

#include <bitset>
bitset<MAXN> bs

Membros: **empty**, **size**, **count**, **to_string**, **to_ulong**, **to_ullong**.
set //Seta todos os elementos para 1.
reset //Seta todos os elementos para 0.
flip(n) //Alterna o bit n.
flip //Alterna todos os bits.
operador » //Shift left.
operador « //Shift right.
operador & //And bit a bit.
operador | //Or bit a bit.
operador ^ //Xor bit a bit.
operador ~ //Not bit a bit.
operador == //Totalmente igual.
operador != //Ao menos um bit é diferente.

2.19 String

#include <string>
string a = "hello";

Membros: **begin**, **end**, **rbegin**, **rend**, **size**, **clear**, **empty operator +** //Concatena string.
operator += ou append(str) //Concatena string.
push_back(c) //Concatena caractere.
push_back(c) //Remove último caractere (C++11).
insert(pos, str) ou insert(it, str) //Concatena caractere.
assign(str) ou assign(n, c) //Atribui string.
erase(pos, len) //Deleta trecho da string.
replace(pos, len, str) //Substitui trecho da string.
swap(str) //Troca conteúdos em $O(1)$.
find(str, pos) //Retorna índice da próxima aparição de str em $O(n)$. Retorna string::npos se não achar.
substr(pos, len) //Retorna substring.

2.20 Algorithm e numeric

#include <algorithm> ou **#include <numeric>**
beg e end podem ser ponteiros para arrays do tipo **T** ou iteradores de uma coleção de container tipo **T**. Quando falarmos em comparador, falamos de funções **bool comp(T a, T b)**, que simulam "menor que". Quando falarmos em evaluadores, falamos em funções **bool eval(T a)**. Quando falarmos em somadores, falamos em funções **T add(T a, T b)**. Todos os ponteiros de funções usados abaixo podem ser codados com lambda functions em C++11.

2.21 Algorithm: Não modificadores

any_of(beg, end, eval) //Retorna se todos os elementos em [beg,end) são evaluados como true pelo evaluador eval.
all_of(beg, end, eval) //Retorna se algum elemento em [beg,end) é evaluado como true pelo evaluador eval.
none_of(beg, end, eval) //Retorna se nenhum elemento em [beg,end) é evaluado como true pelo evaluador eval.
for_each(beg, end, proc) //Executa a função **void proc(T a)** para cada elemento em [beg, end).
count(beg, end, c) //Conta quantos elementos em [beg, end) são iguais a c.
count_if(beg, end, eval) //Conta quantos elementos em [beg, end) são evaluados como true pelo evaluador eval.

2.22 Algorithm: Modificadores

fill(beg, end, c) //Atribui c a todos os elementos em [beg, end).
generate(beg, end, acum) //Atribui a cada posição em [beg,end) o valor retornado por **T acum()** na ordem (usar variáveis globais ou estáticas para valores distintos).
remove(beg, end, c) //Remove todos os elementos em [beg, end) que são iguais a c, retorna o ponteiro para o novo fim de intervalo ou o novo iterador end.
remove_if(beg, end, eval) //Remove todos os elementos em [beg, end) que forem evaluados como true pelo evaluador eval, retorna o ponteiro para o novo fim de intervalo ou novo iterador end.
replace(beg, end, c, d) //Substitui por d todos os elementos em [beg, end) que são iguais a c.
replace_if(beg, end, eval, c) //Substitui por d todos os elementos em [beg, end) que forem evaluados como true pelo evaluador eval.
swap(a, b) //Troca o conteúdo de a e b. Para a maior parte das coleções do C++, é $O(1)$.
reverse(beg, end) //Inverte a ordem em [beg, end).
rotate(beg, beg+i, end) //Rotaciona [beg, end) de forma que o i-ésimo elemento fique em primeiro.
random_shuffle(beg, end) //Aplica permutação aleatória em [beg, end).
unique(beg, end) //Remove todas as duplicatas de elementos consecutivos iguais em [beg, end), retorna o ponteiro para o novo fim de intervalo o novo iterador end.

2.23 Algorithm: Partições

partition(beg, end, eval) //Reordena [beg,end) de forma a que todos os elementos que sejam evaluados como true pelo evaluador eval venham antes dos que sejam evaluados como false. Ordem de cada parte é indefinida.
stable_partition(beg, end, eval) //Mesmo que acima, mas a ordem de cada partição é preservada.

2.24 Algorithm: Ordenação

is_sorted(beg, end) ou **is_sorted(beg, end, comp)** (C++11) //Verifica se [beg, end) está ordenado de acordo com o operador $<$ ou de acordo com o comparador comp.
sort(beg, end) ou **sort(beg, end, comp)** //Ordena [beg, end) de acordo com o operador $<$ ou de acordo com o comparador comp.
stable_sort(beg, end) ou **stable_sort(beg, end, comp)** //Ordena [beg, end) de acordo com o operador $<$ ou de acordo com o comparador comp. Mantém a ordem de elementos iguais.
nth_element(beg, beg+n, beg) ou **nth_element(beg, beg+n, beg, comp)** //Realiza a partição de [beg, end) de forma a que o n-ésimo fique no lugar, os menores fiquem antes e os maiores, depois. *Expected* $O(n)$. Usa o operador $<$ ou o comparador comp.

2.25 Algorithm: Busca binária

lower_bound(beg, end, c) ou **lower_bound(beg, end, c, comp)** //Retorna o ponteiro ou iterador ao primeiro elemento maior que ou igual a c na array ordenada [beg, end) de acordo com o operador $<$ ou de acordo com o comparador comp.
upper_bound(beg, end, c) ou **upper_bound(beg, end, c, comp)** //Retorna o ponteiro ou iterador ao primeiro elemento maior que c na array ordenada [beg, end) de acordo com o operador $<$ ou de acordo com o comparador comp.
binary_search(beg, end, c) ou **binary_search(beg,**

end, c, comp) //Retorna se o elemento c na array ordenada [beg, end) de acordo com o operador $<$ ou de acordo com a função **bool comp(T a, T b)**, que simula "menor que".

2.26 Algorithm: Heap

make_heap(beg, end) ou **make_heap(beg, end, comp)** //Transforma [beg,end) em uma heap de máximo de acordo com o operador $<$ ou de acordo com o comparador **comp**.
push_heap(beg, end, c) ou **push_heap(beg, end, c, comp)** //Adiciona à heap de máximo [beg,end) o elemento c.
pop_heap(beg, end) ou **pop_heap(beg, end, comp)** //Remove da heap de máximo [beg,end) o maior elemento. Joga ele para o final.
sort_heap(beg, end) ou **sort_heap(beg, end, comp)** //Ordena a heap de máximo [beg,end) de forma crescente.

2.27 Algorithm: Máximo e mínimo

max(a,b) //Retorna o maior valor de a e b.
min(a,b) //Retorna o menor valor de a e b.
max_element(beg, end) ou **max_element(beg, end, comp)** //Retorna o elemento máximo em [beg, end) pelo operador $<$ ou pela comparador comp.
min_element(beg, end) ou **min_element(beg, end, comp)** //Retorna o elemento mínimo em [beg, end) pelo operador $<$ ou pela comparador comp..

2.28 Algorithm: Permutações

Use **sort** para obter a permutação inicial!
next_permutation(beg, end) ou **next_permutation (beg, end, comp)** //Reordena [beg, end) para a próxima permutação segundo a ordenação lexicográfica segundo o operador $<$ ou segundo o comparador comp. $O(n)$. Retorna se existe próxima permutação ou não (bool).
prev_permutation(beg, end) ou **prev_permutation (beg, end, comp)** //Reordena [beg, end) para a permutação anterior segundo a ordenação lexicográfica segundo o operador $<$ ou segundo o comparador comp. $O(n)$. Retorna se existe permutação anterior ou não (bool).

2.29 Numeric: Acumuladores

accumulate(beg, end, st) ou **accumulate(beg, end, st, add)** //Soma todos os elementos em [beg, end) a partir de um valor inicial st usando o operador + ou o somador add.
partial_sum(beg, end) ou **partial_sum(beg, end, add)** //Transforma [beg, end) em sua array de somas parciais usando o operador + ou o somador add. **partial_sum(beg, end, st)** ou **partial_sum(beg, end, st, add)** //Coloca na array iniciando em st a array de somas parciais de [beg, end) usando o operador + ou o somador add.

2.30 Functional

#**include** <**functional**>
Algumas funções binárias úteis, especialmente para as funções acima. Quando falamos em agregar, falamos em funções binárias to tipo **T add(T a, T b)**. Quando falamos em comparadores, falamos em funções binárias do tipo **bool comp(T a, T b)**. Quando falamos em transformações, falamos em funções unárias do tipo **T t(T a)**.
plus<**T**>() //Agregador pelo + do tipo T.
minus<**T**>() //Agregador pelo - do tipo T.
multiplies<**T**>() //Agregador operador * do tipo T.
divides<**T**>() //Agregador pelo / do tipo T.
modulus<**T**>() //Agregador pelo % do tipo T.
negate<**T**>() //Transformador pelo - do tipo T.
equal_to<**T**>() //Comparador pelo == do tipo T.
not_equal_to<**T**>() //Comparador pelo != do tipo T.
greater<**T**>() //Comparador pelo $>$ do tipo T.
less<**T**>() //Comparador pelo $<$ do tipo T.
greater_equal<**T**>() //Comparador pelo $>=$ do tipo T.
less_equal<**T**>() //Comparador pelo $<=$ do tipo T.
logical_and<**T**>() //Comparador pelo && do tipo T.
logical_or<**T**>() //Comparador pelo || do tipo T.
bind1st(f, k) //Transforma a função binária em unária fixando o primeiro argumento a k.
bind2nd(f, k) //Transforma a função binária em unária fixando o segundo argumento a k.

Capítulo 3

Estruturas de dados

.

3.1 Max-Queue

```
class MaxQueue {
   list<ii> q, l;
   int cnt = 0;
public:
   MaxQueue() : cnt(0) {}
   void push(int x) {
      ii cur = ii(x, cnt++);
      while(!l.empty() && l.back() <= cur) l.
          pop_back();
      q.push_back(cur);
      l.push_back(cur);
   }
   int front() { return q.front().first; }
   void pop() {
      if (q.front().second == l.front().second) l.
          pop_front();
      q.pop_front();
   }
   int max() { return l.front().first; }
   int size() { return q.size(); }
};
```

3.2 Union-Find

```
class UnionFind {
private:
   vector<int> parent, rank;
public:
   UnionFind(int N) {
      rank.assign(N+1, 0);
      parent.assign(N+1, 0);
      for (int i = 0; i <= N; i++) parent[i] = i;
   }
   int find(int i) {
      while(i != parent[i]) i = parent[i];
      return i;
   }
   bool isSameSet(int i, int j) {
      return find(i) == find(j);
   }
   void unionSet (int i, int j) {
      if (isSameSet(i, j)) return;
      int x = find(i), y = find(j);
      if (rank[x] > rank[y]) parent[y] = x;
      else {
         parent[x] = y;
         if (rank[x] == rank[y]) rank[y]++;
      }
   }
};
```

3.3 Binary Indexed Tree / Fenwick Tree

Resolve queries do tipo RSQ de 1 a n (1-indexed) em $O(\log n)$. Update pontual em $O(\log n)$. Não fazer update com $i = 0$.

```
const int neutral = 0;
#define comp(a, b) ((a)+(b))

class FenwickTree {
private:
   vector<int> ft;
public:
   FenwickTree(int n) { ft.assign(n + 1, 0); }
   int rsq(int i) { // returns RSQ(1, i)
      int sum = neutral;
      for(; i; i -= (i & -i))
         sum = comp(sum, ft[i]);
      return sum;
   }
   int rsq(int i, int j) {
      return rsq(j) - rsq(i - 1);
   }
   void update(int i, int v) {
      for(; i < (int)ft.size(); i += (i & -i))
         ft[i] = comp(v, ft[i]);
   }
};
```

3.4 Binary Indexed Tree / Fenwick Tree com range updates e queries

Resolve queries do tipo RSQ de i a j (1-indexed) em $O(\log n)$. Range updates ($a[i...j] += v$) em $O(\log n)$.

```
class FenwickTree {
private:
   vector<int> ft1, ft2;
   int rsq(vector<int> & ft, int i) {
      int sum = 0;
      for(; i; i -= (i & -i)) sum += ft[i];
      return sum;
   }
   void update(vector<int> & ft, int i, int v) {
      for(; i < (int)ft.size(); i += (i & -i))
         ft[i] += v;
   }
public:
   FenwickTree(int n) {
      ft1.assign(n + 1, 0); //1-indexed
      ft2.assign(n + 1, 0); //1-indexed
   }
   void update(int i, int j, int v) {
      update(ft1, i, v);
      update(ft1, j+1, -v);
      update(ft2, i, v*(i-1));
      update(ft2, j+1, -v*j);
   }
   int rsq(int i) {
      return rsq(ft1, i)*i - rsq(ft2, i);
   }
   int rsq(int i, int j) {
      return rsq(j) - rsq(i-1);
   }
};
```

3.5 2D Binary Indexed Tree / Fenwick Tree

```
const int neutral = 0;
#define comp(a, b) ((a)+(b))

class FenwickTree2D {
private:
   vector< vector<int> > ft;
public:
   FenwickTree2D(int n, int m) {
      ft.assign(n + 1, vector<int>(m + 1, 0)); //1-indexed
   }
   int rsq(int i, int j) { // returns RSQ((1,1), (i, j))
      int sum = 0, _j = j;
      while(i > 0) { j = _j;
         while(j > 0) {
            sum = comp(sum, ft[i][j]);
            j -= (j & -j);
         }
         i -= (i & -i);
      }
      return sum;
   }
   void update(int i, int j, int v) {
      int _j = j;
      while(i < (int)ft.size()) { j = _j;
         while(j < (int)ft[i].size()) {
            ft[i][j] = comp(v, ft[i][j]);
            j += (j & -j);
         }
         i += (i & -i);
      }
   }
};
```

3.6 Segment Tree

Árvore dos segmentos em 1D, construtor para construção com array em $O(n)$. Queries e updates em $O(\log n)$, memória $O(n)$. Indexado em 0. O update substitui o valor no local, não executa *comp*.

```
const int neutral = 0;
#define comp(a, b) ((a)+(b))

class SegmentTree {
   vector<int> a;
   int n;
public:
   SegmentTree(int* st, int* en) {
      int sz = int(en-st);
      for (n = 1; n < sz; n <<= 1);
      a.assign(n << 1, neutral);
      for(int i=0; i<sz; i++) a[i+n] = st[i];
      for(int i=n+sz-1; i>1; i--)
         a[i>>1] = comp(a[i>>1], a[i]);
   }
   void update(int i, int x) {
      a[i += n] = x; //substitui
      for (i >>= 1; i; i >>= 1)
         a[i] = comp(a[i<<1], a[1+(i<<1)]);
   }
   int query(int l, int r) {
      int ans = neutral;
      for (l+=n, r+=n+1; l<r; l>>=1, r>>=1) {
         if (l & 1) ans = comp(ans, a[l++]);
         if (r & 1) ans = comp(ans, a[--r]);
      }
      return ans;
   }
};
```

3.7 Segment Tree com Lazy Propagation

Árvore dos segmentos em 1D, construtor para construção com array em $O(n)$. Queries e updates em $O(\log n)$, memória $O(n)$. Indexado em 0. O update soma um valor em todos os pontos no intervalo $[a, b]$, mas pode ser modificado para aplicar uma função linear.

```
const int neutral = 0; //comp(x, neutral) = x
#define comp(a, b) ((a)+(b))

class SegmentTree {
private:
   vector<int> st, lazy;
   int size;
#define left(p) (p << 1)
#define right(p) ((p << 1) + 1)
   void build(int p, int l, int r, int* A) {
      if (l == r) { st[p] = A[l]; return; }
      int m = (l + r) / 2;
      build(left(p), l, m, A);
      build(right(p), m+1, r, A);
      st[p] = comp(st[left(p)], st[right(p)]);
   }
   void push(int p, int l, int r) {
      st[p] += (r - l + 1)*lazy[p]; //Caso RSQ
      //st[p] += lazy[p];           //Caso RMQ
      if (l != r) {
         lazy[right(p)] += lazy[p];
         lazy[left(p)] += lazy[p];
      }
      lazy[p] = 0;
   }
   void update(int p, int l, int r, int a, int b,
       int k) {
      push(p, l, r);
      if (a > r || b < l) return;
      else if (l >= a && r <= b) {
         lazy[p] = k; push(p, l, r); return;
      }
      update(left(p), l, (l + r) / 2, a, b, k);
      update(right(p), (l + r) / 2 + 1, r, a, b, k);
      st[p] = comp(st[left(p)], st[right(p)]);
   }
   int query(int p, int l, int r, int a, int b) {
      push(p, l, r);
      if (a > r || b < l) return neutral;
      if (l >= a && r <= b) return st[p];
      int m = (l + r) / 2;
      int p1 = query(left(p), l, m, a, b);
      int p2 = query(right(p), m+1, r, a, b);
      return comp(p1, p2);
   }
public:
   SegmentTree(int* bg, int* en) {
      size = (int)(en - bg);
      st.assign(4 * size, neutral);
      lazy.assign(4 * size, 0);
      build(1, 0, size - 1, bg);
   }
   int query(int a, int b) { return query(1, 0, size
       - 1, a, b); }
   void update(int a, int b, int k) { update(1, 0,
       size - 1, a, b, k); }
};
```

3.8 2D Segment Tree

Árvore de Segmentos 2D $O(q \log^2 n)$ em tempo e memória. Suporta operações as *point-update* e *range-query* de adição. O elemento neutro é definido como 0 pelo padrão do compilador. *st* de nós-y possui a raíz da árvore-x, o de nós-x, o valor.

```
int rs[MAXN], ls[MAXN], st[MAXN], cnt = 0;

class SegmentTree2D {
   int sizex, sizey, v, root;
   int x, y, ix, jx, iy, jy;
   void updatex(int p, int lx, int rx) {
      if (x < lx || rx < x) return;
      st[p] += v;
      if (lx == rx) return;
      if (!rs[p]) rs[p] = ++cnt, ls[p] = ++cnt;
      int mx = (lx + rx) / 2;
      updatex(ls[p], lx, mx);
      updatex(rs[p], mx + 1, rx);
   }
   void updatey(int p, int ly, int ry) {
      if (y < ly || ry < y) return;
      if (!st[p]) st[p] = ++cnt;
      updatex(st[p], 0, sizex);
      if (ly == ry) return;
      if (!rs[p]) rs[p] = ++cnt, ls[p] = ++cnt;
      int my = (ly + ry) / 2;
      updatey(ls[p], ly, my);
      updatey(rs[p], my + 1, ry);
   }
   int queryx(int p, int lx, int rx) {
      if (!p || jx < lx || ix > rx) return 0;
      if (ix <= lx && rx <= jx) return st[p];
      int mx = (lx + rx) / 2;
      return queryx(ls[p], lx, mx) +
         queryx(rs[p], mx + 1, rx);
   }
   int queryy(int p, int ly, int ry) {
      if (!p || jy < ly || iy > ry) return 0;
      if (iy <= ly && ry <= jy) return queryx(st[p],
          0, sizex);
      int my = (ly + ry) / 2;
      return queryy(ls[p], ly, my) +
         queryy(rs[p], my + 1, ry);
   }
public:
   SegmentTree2D(int nx, int ny) : sizex(nx), sizey(
       ny) {
      root = ++cnt;
   }
   void update(int _x, int _y, int _v) {
      x = _x; y = _y; v = _v;
      updatey(root, 0, sizey);
   }
   int query(int _ix, int _jx, int _iy, int _jy) {
      ix = _ix; jx = _jx; iy = _iy; jy = _jy;
      return queryy(root, 0, sizey);
   }
};
```

3.9 Persistent Segment Tree

Segment Tree Persistente. Ao começar a usar a árvore, chamar o construtor com o tamanho exato. *update* retorna o número da nova versão. $MAXS$ deve ser da ordem de $2N + Q \log N$. As versões são indexadas em 0, sendo 0 a versão original.

```
#define MAXS 2000009
const int neutral = 0; //comp(x, neutral) = x
#define comp(a, b) ((a)+(b))

int nds, st[MAXS], ls[MAXS], rs[MAXS];

class PersistentSegmentTree {
private:
   int vroot[MAXS];
   int size, nds, nv;
   int newnode() {
      ls[nds] = rs[nds] = -1;
      st[nds++] = neutral;
      return nds-1;
   }
   void build(int p, int l, int r, int* A) {
      if (l == r) {
         st[p] = A ? A[l] : neutral;
         return;
      }
      ls[p] = newnode();
      rs[p] = newnode();
      int m = (l + r) / 2;
      build(ls[p], l, m, A);
      build(rs[p], m+1, r, A);
      st[p] = comp(st[ls[p]], st[rs[p]]);
   }
   void update(int prv, int p, int l, int r, int i,
    int k) {
      if (i > r || i < l || l > r) return;
      int m = (l + r) / 2;
      if (l == r) st[p] = k;
      else if (i <= m) {
         rs[p] = rs[prv];
         ls[p] = newnode();
         update(ls[prv], ls[p], l, m, i, k);
         st[p] = comp(st[ls[p]], st[rs[p]]);
      }
      else {
         ls[p] = ls[prv];
         rs[p] = newnode();
         update(rs[prv], rs[p], m+1, r, i, k);
         st[p] = comp(st[ls[p]], st[rs[p]]);
      }
   }
   int query(int p, int l, int r, int a, int b) {
      if (a > r || b < l || l > r) return neutral;
      if (l >= a && r <= b) return st[p];
      int p1 = query(ls[p], l, (l + r) / 2, a, b);
      int p2 = query(rs[p], (l + r) / 2 + 1, r, a, b
          );
      return comp(p1, p2);
   }
public:
   PersistentSegmentTree() { size = nds = nv = 0; }
   PersistentSegmentTree(int* begin, int* end) {
      nds = nv = 0; size = (int)(end-begin);
      vroot[nv++] = newnode();
      build(vroot[0], 0, size-1, begin);
   }
   PersistentSegmentTree(int _size) {
      nds = nv = 0; size = _size;
      vroot[nv++] = newnode();
      build(vroot[0], 0, size-1, NULL);
   }
   int query(int a, int b, int v) { return query(
       vroot[v], 0, size-1, a, b); }
   int update(int i, int v, int k) {
      vroot[nv++] = newnode();
      update(vroot[v], vroot[nv-1], 0, size-1, i, k)
          ;
      return nv-1;
   }
   int nver() { return nv; }
};
```

3.10 LiChao Segment Tree

Dado uma array de pontos ordenados $x[]$, implementa as seguintes operações: $insert\_line(m, b)$ insere uma nova reta $y(x) = mx + b$ em $O(\log n)$. $query(i)$ retorna $max(y(x[i]))$ para todas as retas inseridas em $O(\log n)$. Construtor $O(n)$.

```
class LiChao {
   vector<ll> m, b;
   int n, sz; ll *x;
#define gx(i) (i < sz ? x[i] : x[sz-1])
   void update(int t, int l, int r, ll nm, ll nb) {
      ll xl = nm * gx(l) + nb, xr = nm * gx(r) + nb;
      ll yl = m[t] * gx(l) + b[t], yr = m[t] * gx(r)
          + b[t];
      if (yl >= xl && yr >= xr) return;
      if (yl <= xl && yr <= xr) {
         m[t] = nm, b[t] = nb; return;
      }
      int mid = (l + r) / 2;
      update(t<<1, l, mid, nm, nb);
      update(1+(t<<1), mid+1, r, nm, nb);
   }
public:
   LiChao(ll *st, ll *en) : x(st) {
      sz = int(en - st);
      for(n = 1; n < sz; n <<= 1);
      m.assign(2*n, 0); b.assign(2*n, -INF);
   }
   void insert_line(ll nm, ll nb) {
      update(1, 0, n-1, nm, nb);
   }
   ll query(int i) {
      ll ans = -INF;
      for(int t = i+n; t; t >>= 1)
         ans = max(ans, m[t] * x[i] + b[t]);
      return ans;
   }
};
```

3.11 Split-Merge Segment Tree

Implementa um conjunto de *multiset*'s. O construtor recebe n, o maior valor de um elemento de um *multiset*. Pode ser modificada tal que cada *multiset* é uma *SegmentTree*. Suporta as seguintes operações:

- *newSet*(i): inicializa um novo *multiset* apenas com o elemento i e retorna o *id* dele. $O(\log n)$ tempo e espaço.
- *unionSet*($r1, r2$): transfere todos os elementos de $r2$ para $r1$. $r2 := \emptyset$. Amortizado $O(\log n)$ tempo e $O(1)$ espaço.
- *query*(r, a, b): retorna quantos elementos x em r existem tal que $a \leq x \leq b$. $O(\log n)$ tempo e $O(1)$ espaço.
- *size*(r): retorna quantos existem em r. $O(1)$ tempo e espaço.
- *splitSet*(r, k): deixa r com apenas os k primeiros elementos. O restante é jogado em um novo *multiset*. Retorna o *id* do novo *multiset*. r se torna vazio se $k \geq size(r)$. Retorna -1 se $k \leq 0$. $O(\log n)$ tempo e espaço.
- *at*(r, i): retorna o i-ésimo elemento do *multiset* r. 0-indexed. $O(\log n)$ tempo e $O(1)$ espaço.
- *to_vector*(r, v): adiciona os elementos de r em ordem no final de v. $O(n)$ tempo e $O(n)$ espaço.

```
#include <vector>
using namespace std;
#define MAXN 1000009
#define MAXS 5000009

const int neutral = 0;
#define comp(a, b) ((a)+(b))

class SegmentTree {
   int cnt, n, root[MAXN], nroot;
   int sz[MAXS], ls[MAXS], rs[MAXS];

   int newnode(int _s) {
      ls[cnt] = rs[cnt] = -1;
      sz[cnt++] = _s;
      return cnt-1;
   }
   int build(int l, int r, int i) {
      int t = newnode(1);
      if (l == r) return t;
      int m = (l + r) / 2;
      if (i <= m) ls[t] = build(l, m, i);
      else rs[t] = build(m + 1, r, i);
      return t;
   }
   int split(int t1, int k) {
      if (t1 == -1 || sz[t1] <= k) return -1;
      int t2 = newnode(sz[t1] - k);
      sz[t1] = k;
      int sl = ls[t1] == -1 ? 0 : sz[ls[t1]];
      if (k > sl) rs[t2] = split(rs[t1], k - sl);
      else swap(rs[t1], rs[t2]);
      if (k < sl) ls[t2] = split(ls[t1], k);
      return t2;
   }
   int merge(int t1, int t2) {
      if(t1 == -1 || t2 == -1) return t1+t2+1;
      ls[t1] = merge(ls[t1], ls[t2]);
      rs[t1] = merge(rs[t1], rs[t2]);
      sz[t1] = comp(sz[t1], sz[t2]);
      return t1;
   }
   int query(int t, int l, int r, int a, int b) {
      if (t == -1 || l > b || r < a) return neutral;
      if (l >= a && r <= b) return sz[t];
      int m = (l + r) / 2;
      int p1 = query(ls[t], l, m, a, b);
      int p2 = query(rs[t], m+1, r, a, b);
      return comp(p1, p2);
   }
   int at(int t, int l, int r, int i) {
      if (l == r) return l;
      int sl = ls[t] == -1 ? 0 : sz[ls[t]];
      int m = (l + r) / 2;
      if (i < sl) return at(ls[t], l, m, i);
      else return at(rs[t], m+1, r, i - sl);
   }
   void to_vector(int t, int l, int r, vector<int> &
        v) {
      if (t == -1) return;
      int m = (l + r) / 2;
      to_vector(ls[t], l, m, v);
      to_vector(rs[t], m+1, r, v);
      for(int i = 0; i < sz[t] && l == r; i++)
         v.push_back(l);
   }
public:
   SegmentTree() { }
   SegmentTree(int _n) : n(_n), cnt(0), nroot(0) { }
   int newSet(int i) {
      root[nroot++] = build(0, n, i);
      return nroot-1;
   }
   void unionSet(int r1, int r2) {
      root[r1] = merge(root[r1], root[r2]);
      root[r2] = -1;
   }
   int query(int r, int a, int b) { return query(
       root[r], 0, n, a, b); }
   int size(int r) { return root[r] == -1 ? 0 : sz[
       root[r]]; }
   int splitSet(int r, int k) {
      if (k >= size(r)) return -1;
      if (k <= 0) {
         root[nroot++] = root[r];
         root[r] = -1;
      }
      else root[nroot++] = split(root[r], k);
      return nroot-1;
   }
   int at(int r, int i) { return at(root[r], 0, n, i
       ); }
   void to_vector(int r, vector<int> & v) {
      to_vector(root[r], 0, n, v); }
};
```

3.12 Segment Tree Beats

Queries e updates em $O(\log n)$, memória $O(n)$. Indexado em 0. Update $a[i] = min(a[i], k), \forall i \in [a, b]$. Consulta $\sum_{i=a}^{b} a[i]$.

```
class SegmentTreeBeats {
   vector<int> st, lazy;
   vector<int> val[2], cnt[2];
   int size;
#define left(p) (p << 1)
#define right(p) ((p << 1) + 1)
   void update(int u) {
      map<int, int> aux;
      for(int k = 0; k < 2; k++) {
         aux[val[k][left(u)]] += cnt[k][left(u)];
         aux[val[k][right(u)]] += cnt[k][right(u)];
      }
      for(int k = 0; k < 2; k++) {
         val[k][u] = aux.rbegin()->first;
         cnt[k][u] = aux.rbegin()->second;
         aux.erase(aux.rbegin()->first);
      }
      st[u] = st[left(u)] + st[right(u)];
   }
   void push(int u, int l, int r) {
      if (lazy[u] < val[0][u]) {
         st[u] -= cnt[0][u]*val[0][u];
         val[0][u] = lazy[u];
         st[u] += cnt[0][u]*val[0][u];
      }
      if (l < r) {
         lazy[left(u)]=min(lazy[left(u)],lazy[u]);
         lazy[right(u)]=min(lazy[right(u)],lazy[u]);
      }
   }
   void build(int u, int l, int r, int A[]) {
      if (l == r) {
         val[0][u] = A ? A[r] : INF; cnt[0][u] = 1;
         val[1][u] = -INF; cnt[1][u] = 0;
         st[u] = A[r]; return;
      }
      int m = (l + r) / 2;
      build(left(u), l, m, A);
      build(right(u), m+1, r, A);
      update(u);
   }
   void update(int u, int l, int r, int a, int b,
       int k) {
      push(u, l, r);
      if (b < l || r < a || k >= val[0][u]) return;
      if (a <= l && r <= b && val[1][u] < k) {
         lazy[u] = k;
         push(u, l, r); return;
      }
      int m = (l + r) / 2;
      update(left(u), l, m, a, b, k);
      update(right(u), m+1, r, a, b, k);
      update(u);
   }
   int query(int u, int l, int r, int a, int b) {
      push(u, l, r);
      if (b < l || r < a) return 0;
      if (a <= l && r <= b) return st[u];
      int m = (l + r) / 2;
      int p1 = query(left(u), l, m, a, b);
      int p2 = query(right(u), m+1, r, a, b);
      return p1 + p2;
   }
public:
   SegmentTreeBeats(int *bg, int *en) {
      size = int(en - bg);
      st.resize(4 * size);
      for(int k = 0; k < 2; k++) {
         val[k].resize(4 * size);
         cnt[k].resize(4 * size);
      }
      lazy.assign(4 * size, INF);
      build(1, 0, size-1, bg);
   }
   void update(int a, int b, int k) {
      update(1, 0, size-1, a, b, k);
   }
   int query(int a, int b) {
      return query(1, 0, size-1, a, b);
   }
};
```

3.13 Merge Sort Tree

Constrói a árvore de recursão do merge-sort. $O(n \log n)$ em espaço e em tempo de contrução. *query* retorna o número de elementos no trecho $[i, j]$ que estão em $[a, b]$ em $O(\log^2 n)$.

```
class MergeSortTree {
private:
   vector< vector<int> > st;
   int size;
   #define left(p) (p << 1)
   #define right(p) ((p << 1) + 1)
   void build(int p, int l, int r, int* A) { // O(n)
      st[p].resize(r-l+1);
      if (l == r) { st[p][0] = A[l]; return; }
      int pl = left(p), pr = right(p), m = (l+r)/2;
      build(pl , l, m, A);
      build(pr, m+1, r, A);
      merge(st[pl].begin(), st[pl].end(),
         st[pr].begin(), st[pr].end(),
         st[p].begin());
   }
   int query(int p, int l, int r, int i, int j, int
       a, int b) {
      if (j < l || i > r) return 0;
      if (i <= l && j >= r)
         return upper_bound(st[p].begin(), st[p].end
             (), b) -
            lower_bound(st[p].begin(), st[p].end(),
                a);
      int m = (l + r) / 2;
      return query(left(p), l, m, i, j, a, b) +
         query(2*p+1, m+1, r, i, j, a, b);
   }
public:
   MergeSortTree(int* begin, int* end) {
      size = (int)(end-begin);
      st.assign(4*size, vector<int>());
      build(1, 0, size-1, begin);
   }
   int query(int i, int j, int a, int b) {
      return query(1, 0, size-1, i, j, a, b);
   }
};
```

3.14 Fractional Cascading Merge Sort Tree

Merge Sort Tree que resolve queries de contar quantos elementos no trecho $[i, j]$ estão em $[a, b]$ em $O(\log n)$. $li[u][i]$ é o índice do primeiro elemento maior que ou igual a $st[u][i]$ no filho esquerdo de u. $ri[u][i]$, o mesmo para o direito.

```
class MergeSortTree {
private:
   vector< vector<int> > st, li, ri;
   int size;
   #define left(p) (p << 1)
   #define right(p) ((p << 1) + 1)
   void build(int p, int l, int r, int* A) { // O(n)
      st[p].resize(r-l+1);
      if (l == r) { st[p][0] = A[l]; return; }
      int pl = left(p), pr = right(p), m = (l+r)/2;
      vector<int> &vl = st[pl], &vr = st[pr];
      build(pl, l, m, A);
      build(pr, m+1, r, A);
      merge(st[pl].begin(), st[pl].end(),
         st[pr].begin(), st[pr].end(),
         st[p].begin());
      for(int k = 0, i = 0, j = 0; k < r-l+1; k++) {
         while(i < vl.size() && st[p][k] > vl[i])
               i++;
         while(j < vr.size() && st[p][k] > vr[j])
               j++;
         li[p].push_back(i), ri[p].push_back(j);
      }
      li[p].push_back(vl.size());
      ri[p].push_back(vr.size());
   }
   int query(int p, int l, int r, int i, int j, int
       a, int b) {
      if (j < l || i > r) return 0;
      if (i <= l && j >= r) return b-a;
      int m = (l + r) / 2;
      return query(left(p), l, m, i, j,
            li[p][a], li[p][b]) +
         query(right(p), m+1, r, i, j,
            ri[p][a], ri[p][b]);
   }
public:
   MergeSortTree(int* begin, int* end) {
      size = (int)(end-begin);
      st.assign(4*size, vector<int>());
      li.assign(4*size, vector<int>());
      ri.assign(4*size, vector<int>());
      build(1, 0, size-1, begin);
   }
   int query(int i, int j, int a, int b) {
      a = lower_bound(st[1].begin(), st[1].end(), a)
            - st[1].begin();
      b = upper_bound(st[1].begin(), st[1].end(), b)
            - st[1].begin();
      return query(1, 0, size-1, i, j, a, b);
   }
};
```

3.15 Sparse Table

Resolve queries do tipo RMQ $[l, r]$ em $O(1)$. Pré-processamento $O(n \log n)$.

```
#define comp(a, b) min((a),(b))

class SparseTable {
   int st[MAXN][MAXLOGN], sz;
public:
   SparseTable(int* bg, int* en) {
      sz = int(en - bg);
      for(int i=0; i<sz; i++) st[i][0] = bg[i];
      for(int j = 1; 1 << j <= sz; j++)
         for(int i=0; i + (1<<j) <= sz; i++)
            st[i][j] = comp(st[i][j-1], st[i+(1<<(j
                -1))][j-1]);
   }
   int query(int l, int r) {
      int k = (int)floor(log((double)r-l+1) / log
          (2.0)); // 2^k <= (j-i+1)
      return comp(st[l][k], st[r-(1<<k)+1][k]);
   }
};
```

3.16 Pre-Suf Table

Resolve queries de RSQ $[l, r]$ sem usar a operação inversa em $O(1)$. Pré-processamento $O(n \log n)$.

```
const int neutral = 0;
#define comp(a, b) ((a)+(b))
int top[2*MAXN];

class PresufTable {
   vector< vector<int> > f;
public:
   PresufTable(int* st, int* en) {
      int bit = 0, sz = int(en - st), n, k;
      for(n = 1, k = 0; n < sz; n <<= 1, k++);
      f.assign(k, vector<int>(n));
      for(int i = 0; i < n; i++) {
         while((1<<bit) <= i) bit++;
         top[i] = bit;
      }
      for(int i = 0; i < n; i++)
         f[0][i] = i < sz ? st[i] : neutral;
      for(int j=0, len=1; j <= k; j++, len<<=1)
         for(int s = len; s < n; s += (len<<1)) {
            f[j][s] = st[s]; f[j][s-1] = st[s-1];
            for(int i = 1; i < len; i++) {
               f[j][s+i]=comp(f[j][s+i-1],st[s+i]);
               f[j][s-1-i]=comp(f[j][s-i],st[s-1-i])
                   ;
            }
         }
   }
   int query(int l, int r) {
      if (l == r) return f[0][r];
      int i = top[l^r] - 1;
      return comp(f[i][r], f[i][l]);
   }
};
```

3.17 AVL Tree

```
struct node {
   int key, height, size;
   node *left, *right;
   node(int k) {
      key = k; left = right = 0;
      height = size = 1;
   }
};

class AVLtree {
private:
   node* root;
   int size_;
   int height(node* p) {
      return p ? p->height : 0;
   }
   int size(node* p) {
      return p ? p->size : 0;
   }
   int bfactor(node* p) {
      return height(p->right) - height(p->left);
   }
   void fixheight(node* p) {
      int hl = height(p->left);
      int hr = height(p->right);
      p->height = (hl>hr ? hl : hr) + 1;
      p->size = 1 + size(p->left) + size(p->right);
   }
   node* rotateright(node* p) {
      node* q = p->left;
      p->left = q->right;
      q->right = p;
      fixheight(p);
      fixheight(q);
      return q;
   }
   node* rotateleft(node* q) {
      node* p = q->right;
      q->right = p->left;
      p->left = q;
      fixheight(q);
      fixheight(p);
      return p;
   }
   node* balance(node* p) {
      fixheight(p);
      if (bfactor(p) == 2) {
         if (bfactor(p->right)<0)
            p->right = rotateright(p->right);
         return rotateleft(p);
      }
      if (bfactor(p) == -2) {
         if (bfactor(p->left)>0)
            p->left = rotateleft(p->left);
         return rotateright(p);
      }
      return p;
   }
   node* build(node* p, int k) {
      if (!p) return new node(k);
      if (p->key == k) return p;
      else if (k<p->key) p->left = build(p->left, k)
          ;
      else p->right = build(p->right, k);
      return balance(p);
   }
   node* findmin(node* p) {
      return p->left ? findmin(p->left) : p;
   }
   node* removemin(node* p) {
      if (p->left == 0) return p->right;
      p->left = removemin(p->left);
      return balance(p);
   }
   node* remove(node* p, int k) {
      if (!p) return 0;
      if (k < p->key) p->left = remove(p->left, k);
      else if (k > p->key) p->right = remove(p->
          right, k);
      else {
         node* l = p->left;
         node* r = p->right;
         delete p;
         if (!r) return l;
         node* min = findmin(r);
         min->right = removemin(r);
         min->left = l;
         return balance(min);
      }
      return balance(p);
   }
   bool find(node* p, int k) {
      if (!p) return false;
      if (p->key == k) return true;
      else if (k<p->key) return find(p->left, k);
      else return find(p->right, k);
   }
   void del(node* p) {
      if (!p) return;
      del(p->left);
      del(p->right);
      delete p;
   }
   node* nth(node* p, int n) {
      if (!p) return p;
      if (size(p->left) + 1 > n) return nth(p->left,
          n);
      if (size(p->left) + 1 < n) return nth(p->right
          , n - size(p->left) - 1);
      else return p;
   }

public:
   AVLtree() { root = 0; size_ = 0; }
   ~AVLtree() { del(root); }
   bool empty() { return size_ == 0; }
   int size() { return size_; }
   void clear() {
      size_ = 0;
      del(root);
      root = 0;
   }
   void insert(int key) {
      size_++;
      root = build(root, key);
   }
   void erase(int key) {
      size_--;
      root = remove(root, key);
   }
   bool count(int key) {
      return find(root, key);
   }
   int nth_element(int n) {
      node* p = nth(root, n);
      if (p) return p->key;
      else return -1;
   }  //1-indexed
};
```

3.18 Treap / Cartesian Tree

Treap simples, suporta operações da BST (*insert*, *count*, *erase*, *nth_element*). Propriedades:

- É 10 vezes mais lento que Red-Black Tree, só usar se for estritamente necessário.
- Se os valores de y forem os valores de uma array e x a posição, as queries de máximo se tornam queries de LCA.
- IMPORTANTE: *split* separa entre k-1 e k.
- *nth_element* é 1-indexed.
- Não suporta valores repetidos de x.

```
#include <cstdio>
#include <set>
#include <algorithm>
using namespace std;

struct node {
   int x, y, size;
   node *l, *r;
   node(int _x) : x(_x), y(rand()), size(1), l(NULL)
       , r(NULL) {}
};

class Treap {
private:
   node* root;
   int size(node* t) { return t ? t->size : 0; }
   node* refresh(node* t) {
      if (!t) return t;
      t->size = 1 + size(t->l) + size(t->r);
      return t;
   }
   void split(node* &t, int k, node* &a, node* &b) {
      node* aux;
      if (!t) a = b = NULL;
      else if (t->x < k) {
         split(t->r, k, aux, b);
         t->r = aux;
         a = refresh(t);
      }
      else {
         split(t->l, k, a, aux);
         t->l = aux;
         b = refresh(t);
      }
   }
   node* merge(node* a, node* b) {
      if (!a || !b) return a ? a : b;
      if (a->y < b->y) {
         a->r = merge(a->r, b);
         return refresh(a);
      }
      else {
         b->l = merge(a, b->l);
         return refresh(b);
      }
   }
   node* count(node* t, int k) {
      if (!t) return NULL;
      else if (k < t->x) return count(t->l, k);
      else if (k == t->x) return t;
      else return count(t->r, k);
   }
   node* nth(node* t, int n) {
      if (!t) return NULL;
      if (n <= size(t->l)) return nth(t->l, n);
      else if (n == size(t->l) + 1) return t;
      else return nth(t->r, n-size(t->l)-1);
   }
   void del(node* &t) {
      if (!t) return;
      if (t->l) del(t->l);
      if (t->r) del(t->r);
      delete t;
      t = NULL;
   }
public:
   Treap() : root(NULL) { }
   ~Treap() { clear(); }
   void clear() { del(root); }
   int size() { return size(root); }
   bool count(int k) { return count(root, k) != NULL
       ; }
   bool insert(int k) {
      if (count(k)) return false;
      node *a, *b;
      split(root, k, a, b);
      root = merge(merge(a, new node(k)), b);
      return true;
   }
   bool erase(int k) {
      node * f = count(root, k);
      if (!f) return false;
      node *a, *b, *c, *d;
      split(root, k, a, b);
      split(b, k+1, c, d);
      root = merge(a, d);
      delete f;
      return true;
   }
   int nth(int n) {
      node* ans = nth(root, n);
      return ans ? ans->x : -1;
   }
};
```

3.19 Treap / Cartesian Tree implícita

Suporta operações do vector, da SegmentTree e reverse em $O(\log n)$ (*insertAt*, *erase*, *at*, *query*, *reverse*). Propriedades:

- Pode trocar trechos do vetor de lugar com *split* e *merge*.
- IMPORTANTE: *split* separa em árvores de tamanho k e $size - k$.
- *at* é 0-indexed.

```
#include <cstdio>
#include <algorithm>
#define INF (1 << 30)
using namespace std;

struct node{
   int y, v, sum, size;
   bool rev;
   node *l, *r;
   node(int _v) : v(_v), sum(_v), y(rand()),
      size(1), l(NULL), r(NULL), rev(false) {}
};

class ImplicitTreap {
private:
   node* root;
   int size(node* t) { return t ? t->size : 0; }
   int sum(node* t) { return t ? t->sum : 0; }
   node* refresh(node* t) {
      if (t == NULL) return t;
      t->size = 1 + size(t->l) + size(t->r);
      t->sum = t->v + sum(t->l) + sum(t->r);
      if (t->l != NULL) t->l->rev ^= t->rev;
      if (t->r != NULL) t->r->rev ^= t->rev;
      if (t->rev) {
         swap(t->l, t->r);
         t->rev = false;
      }
      return t;
   }
   void split(node* &t, int k, node* &a, node* &b) {
      refresh(t);
      node * aux;
      if (!t) a = b = NULL;
      else if (size(t->l) < k) {
         split(t->r, k-size(t->l)-1, aux, b);
         t->r = aux;
         a = refresh(t);
      }
      else {
         split(t->l, k, a, aux);
         t->l = aux;
         b = refresh(t);
      }
   }
   node* merge(node* a, node* b) {
      refresh(a); refresh(b);
      node* aux;
      if (!a || !b) return a ? a : b;
      if (a->y < b->y) {
         a->r = merge(a->r, b);
         return refresh(a);
      }
      else {
         b->l = merge(a, b->l);
         return refresh(b);
      }
   }
   node* at(node* t, int n) {
      if (!t) return t;
      refresh(t);
      if (n < size(t->l)) return at(t->l, n);
      else if (n == size(t->l)) return t;
      else return at(t->r, n-size(t->l)-1);
   }
   void del(node* &t) {
      if (!t) return;
      if (t->l) del(t->l);
      if (t->r) del(t->r);
      delete t;
      t = NULL;
   }
public:
   ImplicitTreap() : root(NULL) { }
   ~ImplicitTreap() { clear(); }
   void clear() { del(root); }
   int size() { return size(root); }
   bool insertAt(int n, int v) {
      node *a, *b;
      split(root, n, a, b);
      root = merge(merge(a, new node(v)), b);
      return true;
   }
   bool erase(int n) {
      node *a, *b, *c, *d;
      split(root, n, a, b);
      split(b, 1, c, d);
      root = merge(a, d);
      if (c == NULL) return false;
      delete c;
      return true;
   }
   int at(int n) {
      node* ans = at(root, n);
      return ans ? ans->v : -1;
   }
   int query(int l, int r) {
      if (l > r) swap(l, r);
      node *a, *b, *c, *d;
      split(root, l, a, d);
      split(d, r-l+1, b, c);
      int ans = sum(b);
      root = merge(a, merge(b, c));
      return ans;
   }
   void reverse(int l, int r) {
      if (l>r) swap(l, r);
      node *a, *b, *c, *d;
      split(root, l, a, d);
      split(d, r-l+1, b, c);
      if (b != NULL) b->rev ^= 1;
      root = merge(a, merge(b, c));
   }
};
```

3.20 Persistent Randomized BST

Treap implícita confluentemente persistente. Suporta *split* e *merge* de versões. *split* retorna o índice da versão da esquerda.

```
struct node {
   int v, sum, size;
   node *l, *r;
   node(int _v, vector<node*> &nodes) :
      v(_v), sum(_v), size(1), l(NULL), r(NULL) {
      nodes.push_back(this);
   }
};

class RBST {
private:
   vector<node*> root, nodes;
   int size(node* t) { return t ? t->size : 0; }
   int sum(node* t) { return t ? t->sum : 0; }
   node* copy(node* t) {
      node* cpy = new node(t->v, nodes);
      cpy->l = t->l; cpy->r = t->r;
      return refresh(cpy);
   }
   node* refresh(node* t) {
      if (t == NULL) return t;
      t->size = 1 + size(t->l) + size(t->r);
      t->sum = t->v + sum(t->l) + sum(t->r);
      return t;
   }
   void split(node* t, int k, node* &a, node* &b) {
      node *aux;
      if (!t) { a = b = NULL; return; }
      t = copy(t);
      if (size(t->l) < k) {
         split(t->r, k-size(t->l)-1, aux, b);
         t->r = aux;
         a = refresh(t);
      }
      else {
         split(t->l, k, a, aux);
         t->l = aux;
         b = refresh(t);
      }
   }
   node* merge(node* a, node* b) {
      node *aux, *cpy;
      if (!a || !b) return a ? a : b;
      int m = size(a), n = size(b);
      if (rand()%(n+m) < m) {
         a = copy(a);
         a->r = merge(a->r, b);
         return refresh(a);
      }
      else {
         b = copy(b);
         b->l = merge(a, b->l);
         return refresh(b);
      }
   }
   node* at(node* t, int i) {
      if (!t) return t;
      if (i < size(t->l)) return at(t->l, i);
      else if (i == size(t->l)) return t;
      else return at(t->r, i-size(t->l)-1);
   }
public:
   RBST() { clear(); }
   ~RBST() { clear(); }
   void clear() {
      for(int i = 0; i < (int)nodes.size(); i++)
         delete nodes[i];
      root.push_back(NULL);
   }
   int size(int ver) { return size(root[ver]); }
   int insert(int ver, int i, int v) {
      node *a, *b, *tree;
      split(root[ver], i, a, b);
      tree = merge(merge(a, new node(v, nodes)), b);
      root.push_back(tree);
      return root.size() - 1;
   }
   int erase(int ver, int i) {
      node *a, *b, *c, *d;
      split(root[ver], i, a, b);
      split(b, 1, c, d);
      root.push_back(merge(a, d));
      return root.size() - 1;
   }
   int at(int ver, int i) {
      node* ans = at(root[ver], i);
      return ans ? ans->v : -1;
   }
   int query(int ver, int l, int r) {
      if (l > r) swap(l, r);
      node *a, *b, *c, *d;
      split(root[ver], l, a, d);
      split(d, r-l+1, b, c);
      return sum(b);
   }
   int split(int ver, int k) {
      node *a, *b;
      split(root[ver], k, a, b);
      root.push_back(a);
      root.push_back(b);
      return root.size()-2;
   }
   int merge(int ver1, int ver2) {
      root.push_back(merge(root[ver1], root[ver2]));
      return root.size()-1;
   }
};
```

3.21 Deque Recursivo Puramente Funcional

$O(1)$ amortizado em tempo e memória para persistência parcial, $O(\log n)$ para total.

```
struct cont {
   cont *ls, *rs;
   int x;
   cont(int _x, vector<cont*> &gc) :
      x(_x), ls(NULL), rs(NULL) {
      gc.push_back(this);
   }
   cont(cont* l, cont* r, vector<cont*> &gc) :
      x(0), ls(l), rs(r) {
      gc.push_back(this);
   }
};

struct node {
   cont* s[2];
   node* c;
   int sz;
   node(vector<node*> &gc) : c(NULL), sz(0) {
      s[0] = s[1] = NULL;
      gc.push_back(this);
   }
   node* copy(vector<node*> &gc) {
      node* cpy = new node(gc);
      cpy->sz = sz; cpy->c = c;
      cpy->s[0] = s[0]; cpy->s[1] = s[1];
      return cpy;
   }
};

class Deque {
private:
   vector<node*> root, nodes;
   vector<cont*> conts;
   void push(node* &u, int k, cont* x) {
      if (!u) { u = new node(nodes); }
      else u = u->copy(nodes);
      u->sz++;
      if (!u->s[k]) u->s[k] = x;
      else {
         cont* y = u->s[k];
         u->s[k] = NULL;
         if (k) swap(x, y);
         push(u->c, k, new cont(x, y, conts));
      }
   }
   cont* pop(node* &u, int k) {
      u = u->copy(nodes);
      u->sz--;
      cont *x = NULL, *y, *z;
      if (u->s[k]) {
         x = u->s[k];
         u->s[k] = NULL;
      }
      else if (u->c && u->c->sz) {
         z = pop(u->c, k);
         x = z->ls; y = z->rs;
         if (k) swap(x, y);
         u->s[k] = y;
      }
      else if (u->s[k^1]) {
         x = u->s[k^1];
         u->s[k^1] = NULL;
      }
      else u->sz++;
      return x;
   }
   cont* at(const node* u, int i) {
      if (!u) return NULL;
      if (i == 0 && u->s[0]) return u->s[0];
      if (i == u->sz-1 && u->s[1]) return u->s[1];
      if (u->s[0]) i--;
      cont* x = at(u->c, i>>1);
      return i & 1 ? x->rs : x->ls;
   }
public:
   Deque() { root.push_back(new node(nodes)); }
   ~Deque() {
      for(int i = 0; i < (int)nodes.size(); i++)
         delete nodes[i];
      for(int i = 0; i < (int)conts.size(); i++)
         delete conts[i];
   }
   int push(int ver, int x, int k) {
      root.push_back(root[ver]);
      push(root.back(), k, new cont(x, conts));
      return root.size()-1;
   }
   int pop(int ver, int k) {
      root.push_back(root[ver]);
      pop(root.back(), k);
      return root.size()-1;
   }
   int push_back(int ver, int x) { return push(ver,
       x, 1); }
   int push_front(int ver, int x) { return push(ver,
        x, 0); }
   int pop_back(int ver) { return pop(ver, 1); }
   int pop_front(int ver) { return pop(ver, 0); }
   int size(int ver) { return root[ver]->sz; }
   int at(int ver, int i) { return at(root[ver], i)
       ->x; }
   int front(int ver) { return at(root[ver], 0)->x;
       }
   int back(int ver) { return at(root[ver], size(ver
       )-1)->x; }
};
```

3.22 Splay Tree

Árvore de busca binária em que, para todas as operações, rotaciona-se a raíz até o elemento desejado chegar na raíz.

```
struct node {
   int key;
   node *ls, *rs;
   node(int k) : key(k), ls(NULL), rs(NULL) {}
};

class SplayTree {
private:
   node* root;
   node* rotateright(node* p) {
      node* q = p->ls;
      p->ls = q->rs;
      q->rs = p;
      return q;
```

```
    }
    node* rotateleft(node* q) {
        node* p = q->rs;
        q->rs = p->ls;
        p->ls = q;
        return p;
    }
    node* splay(node* p, int key) {
        if (!p || p->key == key) return p;
        if (p->key > key) {
            if (!p->ls) return p;
            if (p->ls->key > key)    {
                p->ls->ls = splay(p->ls->ls, key);
                p = rotateright(p);
            }
            else if (p->ls->key < key) {
                p->ls->rs = splay(p->ls->rs, key);
                if (p->ls->rs)
                    p->ls = rotateleft(p->ls);
            }
            return (!p->ls) ? p : rotateright(p);
        }
        else {
            if (!p->rs) return p;
            if (p->rs->key > key) {
                p->rs->ls = splay(p->rs->ls, key);
                if (p->rs->ls)
                    p->rs = rotateright(p->rs);
            }
            else if (p->rs->key < key) {
                p->rs->rs = splay(p->rs->rs, key);
                p = rotateleft(p);
            }
            return (!p->rs) ? p : rotateleft(p);
        }
    }
    void del(node* &p) {
        if (!p) return;
        del(p->ls); del(p->rs);
        delete p;
        p = NULL;
    }
public:
    SplayTree() : root(NULL) { }
    ~SplayTree() { del(root); }
    bool empty() { return root == NULL; }
    void clear() { del(root); }
    void insert(int key) {
        if (!root) {
            root = new node(key);
            return;
        }
        node* p = splay(root, key);
        if (p->key == key) return;
        root = new node(key);
        if (p->key > key) {
            root->rs = p;
            root->ls = p->ls;
            p->ls = NULL;
        }
        else {
            root->ls = p;
            root->rs = p->rs;
            p->rs = NULL;
        }
    }
    void erase(int key) {
        node* p = splay(root, key);
        if (!p || p->key != key) return;
        if (!p->rs) {
            root = p->ls;
            delete p;
            return;
        }
        node* q = splay(p->rs, key);
        q->ls = p->ls;
        root = q;
        delete p;
    }
    bool count(int key) {
        if (!root) return false;
        root = splay(root, key);
        return root->key == key;
    }
};
```

3.23 Link Cut Tree

Estrutura de dados semelhante a disjoint sets que permite conectar vértices sem pai a algum outro vértice, cortar relação com o pai e queries de raíz da árvore e LCA. Tudo em $O(\log n)$.

```
struct node {
    int sw, id, w;
    node *par, *ppar, *ls, *rs;
    node() {
        par = ppar = ls = rs = NULL;
        w = sw = INF;
    }
};

class LinkCutTree {
    vector<node> lct;
    int sum(node *p) { return p ? p->sw : 0; }
    void refresh(node* p) {
        p->sw = p->w + sum(p->ls) + sum(p->rs);
    }
    void rotateright(node* p) {
        node *q, *r;
        q = p->par, r = q->par;
        if (q->ls = p->rs) q->ls->par = q;
        p->rs = q, q->par = p;
        if (p->par = r) {
            if (q == r->ls) r->ls = p;
            else r->rs = p;
        }
        p->ppar = q->ppar;
        q->ppar = NULL;
        refresh(q);
    }
    void rotateleft(node* p) {
        node *q, *r;
        q = p->par, r = q->par;
        if (q->rs = p->ls) q->rs->par = q;
        p->ls = q, q->par = p;
        if (p->par = r) {
            if (q == r->ls) r->ls = p;
            else r->rs = p;
        }
        p->ppar = q->ppar;
        q->ppar = 0;
        refresh(q);
    }
    void splay(node* p) {
        node *q, *r;
        while (p->par != NULL) {
```

```
            q = p->par;
            if (q->par == NULL) {
               if (p == q->ls) rotateright(p);
               else rotateleft(p);
               continue;
            }
            r = q->par;
            if (q == r->ls) {
               if (p == q->ls) rotateright(q),
                  rotateright(p);
               else rotateleft(p), rotateright(p);
            }
            else {
               if (p == q->rs) rotateleft(q),
                  rotateleft(p);
               else rotateright(p), rotateleft(p);
            }
         }
         refresh(p);
      }
      node* access(node* p) {
         splay(p);
         if (p->rs != NULL) {
            p->rs->ppar = p; p->rs->par = NULL;
            p->rs = NULL; refresh(p);
         }
         node* last = p;
         while (p->ppar != NULL) {
            node* q = last = p->ppar;
            splay(q);
            if (q->rs != NULL) {
               q->rs->ppar = q;
               q->rs->par = NULL;
            }
            q->rs = p; p->par = q;
            p->ppar = NULL;
            refresh(q); splay(p);
         }
         return last;
      }
   public:
      LinkCutTree(int n = 0) {
         lct.resize(n + 1);
         for(int i = 0; i <= n; i++) {
            lct[i].id = i;
            refresh(&lct[i]);
         }
      }
      void link(int u, int v, int w = 1) {
         //u becomes child of v
         node *p = &lct[u], *q = &lct[v];
         access(p); access(q);
         p->ls = q; q->par = p; p->w = w;
         refresh(p);
      }
      void cut(int u) {
         node* p = &lct[u]; access(p);
         p->ls->par = NULL; p->ls = NULL;
         refresh(p);
      }
      int findroot(int u) {
         node* p = &lct[u]; access(p);
         while (p->ls) p = p->ls;
         splay(p);
         return p->id;
      }
      bool IsSameTree(int u, int v) {
         return findroot(u) == findroot(v);
      }
      int depth(int u) {
         access(&lct[u]);
         return lct[u].sw - lct[u].w;
      }
      int LCA(int u, int v) {
         access(&lct[u]);
         return access(&lct[v])->id;
      }
      int dist(int u, int v) {
         if (!IsSameTree(u, v)) return INF;
         node* lca = &lct[LCA(u, v)];
         int ans = 0;
         access(&lct[u]); splay(lca);
         ans += sum(lca->rs);
         access(&lct[v]); splay(lca);
         ans += sum(lca->rs);
         return ans;
      }
};
```

3.24 Link Cut Tree não direcionada

Semelhante a *LinkCutTree*, porém permite o *link* e o *cut* de quaisquer vértices de forma não direcionada. Tudo em $O(\log^2 n)$.

```
struct node { ... };
class LinkCutTree { ... };

class UndirectedLinkCutTree {
   LinkCutTree lct;
   vector<int> par;

   void invert(int u) {
      if (par[u] == -1) return;
      int v = par[u];
      invert(v);
      lct.cut(u); par[u] = -1;
      lct.link(v, u); par[v] = u;
   }
public:
   UndirectedLinkCutTree(int n = 0) {
      lct = LinkCutTree(n);
      par.assign(n+1, -1);
   }
   void link(int u, int v) {
      if (lct.depth(u) < lct.depth(v)) {
         invert(u);
         lct.link(u, v); par[u] = v;
      }
      else {
         invert(v);
         lct.link(v, u); par[v] = u;
      }
   }
   void cut(int u, int v) {
      if (par[v] == u) u = v;
      lct.cut(u); par[u] = -1;
   }
   bool IsSameTree(int u, int v) {
      return lct.IsSameTree(u, v);
   }
};
```

3.25 Wavelet Tree

Constrói a Wavelet Tree em $O(n \log \sigma + \sigma)$ em espaço e em tempo de contrução. Todas as operações são indexadas em 0. Ela supõe que todos os elementos da array estão em $[0, \sigma]$. Usar compressão para que $O(\sigma) = O(n)$. Todas as operações são independentes, não precisa codar se não for usar. Esta árvore é mais rápida de codar e de executar para resolver do que a Merge Sort Tree.

Operações:

- $rank(i, j, q)$: Acha quantas vezes um elemento q aparece em $[i, j]$ em $O(\log \sigma)$.
- $quantile(i, j, k)$: Acha o k-ésimo elemento em $[i, j]$ em $O(\log \sigma)$.
- $range(i, j, a, b)$: acha quantos elementos em $[i, j]$ estão contidos em $[a, b]$ $(a \leq arr[k] \leq b, i \leq k \leq j)$ em $O(\log \sigma)$.
- $swap(i)$: troca os elementos na posição i e $i + 1$ em $O(\log \sigma)$.

```
#include <vector>
#include <algorithm>
using namespace std;

int Lcmp;
bool less(int i) { return i <= Lcmp; }

class WaveletTree {
private:
   vector<vector<int> > ml;
   vector<int> arr;
   int sig, size;
#define left(p) (p << 1)
#define right(p) ((p << 1) + 1)

   void build(int u, int l, int r, int lo, int hi,
       int* A) {
      if (lo == hi) return;
      int mid = (lo + hi) / 2;
      Lcmp = mid;
      ml[u].reserve(r-l+2);
      ml[u].push_back(0);
      for (int i=l; i<=r; i++) {
         ml[u].push_back(ml[u].back() + (A[i]<=Lcmp)
             );
      }
      int p = (int)(stable_partition(A+l, A+r+1,
          less) - A);
      build(left(u), l, p-1, lo, mid, A);
      build(right(u), p, r, mid+1, hi, A);
   }

   int rank(int u, int lo, int hi, int q, int i) {
      if (lo == hi) return i;
      int mid = (lo + hi) / 2, ri = ml[u][i];
      if (q <= mid) return rank(left(u), lo, mid, q,
          ri);
      else return rank(right(u), mid+1, hi, q, i -
          ri);
   }

   int quantile(int u, int lo, int hi, int i, int j,
       int k) {
      if (lo == hi) return lo;
      int mid = (lo + hi) / 2;
      int ri = ml[u][i-1], rj = ml[u][j], c = rj -
          ri;
      if (k <= c) return quantile(left(u), lo, mid,
          ri+1, rj, k);
      else return quantile(right(u), mid+1, hi, i-ri
          , j-rj, k-c);
   }

   int range(int u, int lo, int hi, int i, int j,
       int a, int b) {
      if (lo > b || hi < a) return 0;
      if (b >= hi && lo >= a) return j-i;
      int mid = (lo + hi) / 2;
      int ri = ml[u][i], rj = ml[u][j];
      int c1 = range(left(u), lo, mid, ri, rj, a, b)
          ;
      int c2 = range(right(u), mid+1, hi, i-ri, j-rj
          , a, b);
      return c1 + c2;
   }

   void swap(int u, int lo, int hi, int v1, int v2,
       int i) {
      if (lo == hi) return;
      int mid = (lo + hi) / 2;
      if (v1 <= mid) {
         if (v2 > mid) ml[u][i]--;
         else swap(left(u), lo, mid, v1, v2, ml[u][i
             ]);
      }
      else {
         if (v2 <= mid) ml[u][i]++;
         else swap(right(u), mid+1, hi, v1, v2, i-ml
             [u][i]);
      }
   }

public:
   WaveletTree() {}
   WaveletTree(int* begin, int* end, int _sig) {
      sig = _sig;
      size = (int)(end-begin);
      ml.resize(4*size);
      arr = vector<int>(begin, end);
      build(1, 0, size-1, 0, sig, &arr[0]);
      arr = vector<int>(begin, end);
   }

   int rank(int i, int q) { return rank(1, 0, sig, q
       , i+1); }
   int rank(int i, int j, int q) { return rank(j, q)
       - rank(i-1, q); }
   int quantile(int i, int j, int k) { return
       quantile(1, 0, sig, i+1, j+1, k); }
   int range(int i, int j, int a, int b) { return
       range(1, 0, sig, i, j+1, a, b); }
   void swap(int i) {
      if (i >= size-1) return;
      swap(1, 0, sig, arr[i], arr[i+1], i+1);
      std::swap(arr[i], arr[i+1]);
   }
};
```

3.26 Heavy-Light Decomposition

Decomposição Heavy-light de uma árvore em $O(n)$ com uma única estrutura de dados para queries. Query de LCA em $O(\log n)$. *adjList* é modificado de forma a que a maior subárvore esteja na posição 0, dessa forma cadeias e subárvores são trechos contínuos na pré-ordem. $up[u]$ é o nó mais alto da cadeia de u. $in[u]$ é o índice de u na pré-ordem. $[in[u], out[u]]$ é a subárvore de u. $[in[up[u]], in[u]]$ é a cadeia para cima de u.

```
vector<int> adjList[MAXN];
int in[MAXN], out[MAXN], up[MAXN];
int par[MAXN], pre[MAXN], sz[MAXN];

void dfssize(int u) {
   int s = sz[u] = 0;
   for(int i = 0; i < (int)adjList[u].size(); i++) {
      int &v = adjList[u][i]; //don't forget &
      if (v == par[u]) continue;
      par[v] = u;
      dfssize(v);
      s += sz[v];
      if(sz[v] > sz[adjList[u][0]])
         swap(v, adjList[u][0]);
   }
   sz[u] = 1 + s;
}

void dfshld(int u, int &t) {
   pre[t] = u;
   in[u] = t++;
   for(int i = 0; i < (int)adjList[u].size(); i++) {
      int v = adjList[u][i];
      if (v == par[u]) continue;
      up[v] = (i == 0 ? up[u] : v);
      dfshld(v, t);
   }
   out[u] = t-1;
}

void hld(int root) {
   par[root] = -1;
   dfssize(root);
   int t = 0;
   up[root] = root;
   dfshld(root, t);
   //create DS over values sorted by array pre
}

int LCA(int u, int v) {
   while(up[u] != up[v]) {
      if (in[u] > in[v]) u = par[up[u]];
      //query DS for subarray [in[up[u]], in[u]]
      else v = par[up[v]];
      //query DS for subarray [in[up[v]], in[v]]
   }
   return in[u] > in[v] ? v : u;
   //query DS for subarray [in[v], in[u]] or [in[u],
      in[v]]
}
```

3.27 Centroid Decomposition

Realiza a decomposição em $O(n \log n)$ e retorna a raíz da decomposição. $csons[i]$ são os filhos do i-ésimo nó segundo a decomposição. $par[i]$ é o pai do i-ésimo nó segundo a decomposição. $clevel[i]$ é a profundidade do i-ésimo nó na decomposição, iniciando em 0. $csize[i]$ no final do algoritmo contém o tamanho da subárvore de centróides com raíz i.
CUIDADO: o pai da raiz da árvore centróide é ele mesmo.

```
int clevel[MAXN], cpar[MAXN], csize[MAXN];
vector<int> csons[MAXN];
vector<ii> adjList[MAXN];
int N;

int subsize(int u, int p) {
   csize[u]=1;
   for(int i=0; i<(int)adjList[u].size(); i++) {
      int v = adjList[u][i].second;
      if (v != p && clevel[v] < 0)
         csize[u] += subsize(v, u);
   }
   return csize[u];
}
int findcentroid(int u, int p, int nn) {
   for(int i=0; i<(int)adjList[u].size(); i++) {
      int v = adjList[u][i].second;
      if (v != p && clevel[v] < 0 && csize[v] > nn
          /2)
         return findcentroid(v, u, nn);
   }
   return u;
}
void computedistance(int u, int p, int lvl, int d) {
   cdist[lvl][u] = d;
   for(int i=0; i < (int)adjList[u].size(); i++) {
      int v = adjList[u][i].second;
        int w = adjList[u][i].first;
      if (clevel[v] < 0 && v != p) {
         computedistance(v, u, lvl, d+w);
      }
   }
}
int decompose(int root, int par) {
   subsize(root, -1);
   int u = findcentroid(root, -1, csize[root]);
   cpar[u] = par;
   clevel[u] = par >= 0 ? clevel[par]+1 : 0;
    computedistance(u, -1, clevel[u], 0); //Optional
   csize[u] = 1;
   for(int i=0; i<(int)adjList[u].size(); i++) {
      int v = adjList[u][i].second;
      if (v != par && clevel[v] < 0) {
         v = decompose(v, u);
         csons[u].push_back(v);
         csize[u] += csize[v];
      }
   }
   return u;
}
int centroiddecomposition(int root) {
   memset(&clevel, -1, sizeof clevel);
   for(int i=0; i<=N; i++) csons[i].clear();
   return decompose(root, -1);
}
```

3.28 Interval Tree

Implementa a árvore de intervalos com *set*. *get* faz o corte dos intervalos de acordo com $[l, r]$, possivelmente realiza uma atualização e retorna a array com os intervalos contidos em $[l, r]$.

```
#define MAXN 200009

struct node {
   int l, r, x;
   node(int _l, int _r, int _x = 0) : l(_l), r(_r),
       x(_x) { }
   void update(int dx) { x = min(x+dx, 2); }
};

bool operator < (node a, node b) {
   if (a.l != b.l) return a.l < b.l;
   return a.r < b.r;
}

class IntervalTree {
   set<node> tree;
   void split(int i) {
      set<node>::iterator it = --tree.upper_bound(
          node(i, 0));
      node t = *it;
      if (t.l == i) return;
      tree.erase(it);
      tree.insert(node(t.l, i-1, t.x));
      tree.insert(node(i, t.r, t.x));
   }
public:
   IntervalTree() { tree.insert(node(0, MAXN, 0)); }
   vector<node> get(int l, int r, bool update, int
       dx) {
      split(l); split(r+1);
      set<node>::iterator it = tree.lower_bound(node
          (l, 0));
      vector<node> q;
      while(it != tree.end() && it->l <= r) {
         node t = *it;
         it = tree.erase(it);
         if (update) t.update(dx);
         if (q.empty() || q.back().x != t.x) q.
             push_back(t);
         else q.back().r = t.r;
      }
      for(int i = 0; i < int(q.size()); i++)
         tree.insert(q[i]);
      return q;
   }
};
```

3.29 Xor Trie

Implementa as seguintes operações, $O(n \log S)$ de memória, $S = 2^{digits}$, construtor recebe *digits*. *insert* insere um elemento menor que S, $O(\log S)$. *search* retorna o elemento que maximiza o xor com *num* e é menor que *limit*, $O(\log S)$. A adição de tamanho e de lazy propagation pode adicionar queries de remoção durante busca e xor em todos os elementos.

```
#define MAXS 60000009
int l[MAXS], r[MAXS], cnt = 0;

class XorTrie {
   int digits = 0, root, ans, limit;
   int newnode() {
      l[cnt] = r[cnt] = -1; cnt++;
      return cnt-1;
   }
   bool search(int u, int h, int num, int cur) {
      if (u == -1 || cur > limit) return false;
      if (h == -1) { ans = cur; return true; }
      if (num & (1<<h)) {
         if (search(l[u], h-1, num, cur)) return
             true;
         if (search(r[u], h-1, num, cur | (1<<h)))
             return true;
      }
      else {
         if (search(r[u], h-1, num, cur | (1<<h)))
             return true;
         if (search(l[u], h-1, num, cur)) return
             true;
      }
      return false;
   }
public:
   XorTrie(int _digits = 20) : digits(_digits) {
      root = newnode();
   }
   void insert(int num) {
      int u = root;
      for(int i = digits-1; i >= 0; i--) {
         if (num & (1<<i)) {
            if (r[u] == -1) r[u] = newnode();
            u = r[u];
         }
         else {
            if (l[u] == -1) l[u] = newnode();
            u = l[u];
         }
      }
   }
   int search(int _limit, int num) {
      limit = _limit; ans = -1;
      if (!search(root, digits-1, num, 0))
         return -1;
      return ans;
   }
};
```

3.30 Convex Hull Trick

Para queries de mínimo, sete $maxCH = false$ e insira retas do tipo $y = mx + n$ na ordem decrescente de m. Para queries de máximo, sete $maxCH = true$ e insira na ordem crescente de m. *query* resolve queries de $max(y(x))$ ou $min(y(x))$ para todas as retas inseridas em tempo $O(\log n)$. *query_q* resolve em $O(1)$ se o x for crescente, $O(n)$ caso contrário.

```
#include <vector>
#define INF 0x3f3f3f3f
using namespace std;

typedef long long int ll;

class CHTrick {
private:
   vector<ll> m, n;
   vector<double> p;
   bool maxCH; int i;
public:
   CHTrick(bool mxch) { maxCH = mxch; i = 0; }
   void clear() { m.clear(); n.clear(); p.clear(); }
   double inter(double nm, double nn, double lm,
       double ln) {
      return (ln - nn) / (nm - lm);
   }
   void push(ll nm, ll nn) {
      while (!p.empty()) {
         if (nm == m.back() && maxCH && nn <= n.back
             ()) return;
         if (nm == m.back() && !maxCH && nn >= n.
             back()) return;
         double x = inter(nm, nn, m.back(), n.back()
             );
         if (nm != m.back() && p.back() < x) break;
         m.pop_back(); n.pop_back(); p.pop_back();
      }
      p.push_back(p.empty() ? -INF : inter(nm, nn, m
          .back(), n.back()));
      m.push_back(nm); n.push_back(nn);
      if (i >= p.size()) i = p.size()-1;
   }
   ll query(ll x) {
      if (p.empty()) return (maxCH ? -1 : 1)*INF;
      ll r = p.size() - 1, l = 0, mid;
      if (x >= p[r]) return m[r] * x + n[r];
      while (r > l + 1) {
         mid = (r + l) / 2;
         if (x < p[mid]) r = mid;
         else l = mid;
      }
      return m[l] * x + n[l];
   }
   ll query_q(ll x) {
      while (p[i] > x) i--;
      while(i+1 < p.size() && p[i+1] <= x) i++;
      return m[i] * x + n[i];
   }
};
```

3.31 Convex Hull Trick dinâmico e máximo produto interno

insert_line insere uma linha $y(x) = mx + b$ em qualquer ordem em $O(\log^2 n)$. *query*(a, b) retorna a linha que maximiza y(a/b) (otimizado para resolver frações) em $O(\log^2 n)$. Nessa implementação é necessário C++11. *query* retorna o valor máximo de $ax + by$ para um conjunto de pontos (x, y) em $O(\log^2 n)$. Preencher *maxhull* com todos (x, y), *minhull* com $(-x, -y)$, *maxx* com a máxima corrdenada x e *minx* com a mínima.

```
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;

ll nu, de;
struct line {
   ll m, b; bool is_query;
   mutable function<const line *()> succ;
   line(ll _m, ll _b, bool iq = false) :
      m(_m), b(_b), is_query(iq) {}
   bool operator<(const line &o) const {
      if (!o.is_query) return m < o.m;
      const line *s = succ();
      if (!s) return 0;
      return (b - s->b) * de < (s->m - m) * nu;
   }
};

struct DynamicHull : public multiset<line> {
   bool bad(iterator y) {
      auto z = next(y);
      if (y == begin()) {
         if (z == end()) return 0;
         return y->m == z->m && y->b <= z->b;
      }
      auto x = prev(y);
      if (z == end())
         return y->m == x->m && y->b <= x->b;
      return (x->b - y->b) * (z->m - y->m) >= (y->b
          - z->b) * (y->m - x->m);
   }
   void insert_line(ll m, ll b) {
      auto y = insert(line(m, b));
      y->succ = [=] { return next(y) == end() ? 0 :
          &*next(y); };
      if (bad(y)) { erase(y); return; }
      while (next(y) != end() && bad(next(y)))
         erase(next(y));
      while (y != begin() && bad(prev(y)))
         erase(prev(y));
   }
   line query(ll a, ll b) {
      if (b < 0) a = -a, b = -b;
      nu = a; de = b;
      return *lower_bound(line(0, 0, true));
   }
};

ll minx = INF, maxx = -INF;
DynamicHull maxhull, minhull;
ll query(ll a, ll b) {
   if (b == 0) return a*(a > 0 ? maxx : minx);
   if (b > 0) { line l = maxhull.query(a, b);
      return l.m*a + l.b*b;
   }
   else { line l = minhull.query(a, b);
      return -a*l.m + -b*l.b;
   }
}
```

3.32 Van Emde Boas Tree

Árvore que suporta elementos em $[0, \sigma), \sigma = 2^k$ e resolve consultas de conjunto, sucessor e antecessor em $O(\log\log\sigma)$.

```
class VEB {
private:
   VEB* ups;
   vector<VEB*> los;
   int minx, maxx, k, tk;
#define upper(x) ((x)>>tk)
#define lower(x) ((x)&((1<<tk)-1))
#define index(x, y) (((x)<<tk)^(y))
public:
   VEB(int _k = 20) : k(_k), tk(k>>1) {
      minx = -1, maxx = -1;
      ups = NULL;
      if (k == 1) return;
      ups = new VEB(k-tk);
      los.resize(1<<(k-tk), NULL);
   }
   ~VEB() {
      delete ups;
      for(int i = 0; i < (int)los.size(); i++)
         if (los[i]) delete los[i];
   }
   int min() { return minx; }
   int max() { return maxx; }
   bool count(int x) {
      int up = upper(x), lo = lower(x);
      if (x == minx || x == maxx) return true;
      if (k == 1 || !los[up]) return false;
      return los[up]->count(lo);
   }
   int succ(int x) {
      int up = upper(x), lo = lower(x);
      if (k == 1) return x == 0 && maxx == 1 ? 1 :
          -1;
      if (minx != -1 && x < minx) return minx;
      int mlo = los[up] ? los[up]->max() : -1;
      if (mlo != -1 && lo < mlo)
         return index(up, los[up]->succ(lo));
      up = ups->succ(up);
      if (up == -1) return -1;
      return index(up, los[up]->min());
   }
   int precc(int x) {
      int up = upper(x), lo = lower(x);
      if (k == 1) return x == 1 && minx == 0 ? 0 :
          -1;
      if (maxx != -1 && x > maxx) return maxx;
      int mlo = los[up] ? los[up]->min() : -1;
      if (mlo != -1 && lo > mlo)
         return index(up, los[up]->precc(lo));
      up = ups->precc(up);
      if (up == -1)
         return minx != -1 && x > minx ? minx : -1;
      return index(up, los[up]->max());
   }
   void insert(int x) {
      if (minx == -1) {
         minx = maxx = x; return;
      }
      if (x < minx) swap(x, minx);
      if (x > maxx) maxx = x;
      if (k == 1 || x == minx) return;
      int up = upper(x), lo = lower(x);
      if (!los[up] || los[up]->min() == -1) {
         ups->insert(up);
         los[up] = new VEB(tk);
      }
      los[up]->insert(lo);
   }
   int removemin() {
      if (minx == -1) return -1;
      int ans = minx;
      if (minx == maxx) {
         minx = maxx = -1; return ans;
      }
      if (k == 1) { minx = maxx; return ans; }
      int up = ups->min();
      minx = index(up, los[up]->removemin());
      if (los[up]->min() == -1) ups->removemin();
      return ans;
   }
   void erase(int x) {
      if (x == -1) return;
      int up = upper(x), lo = lower(x);
      if (x == minx) removemin();
      else if (k == 1 && x == maxx) maxx = minx;
      else {
         los[up]->erase(lo);
         if (los[up]->min() == -1) ups->erase(up);
         if (x == maxx && los[up]->max() != -1)
            maxx = index(up, los[up]->max());
         else if (x == maxx) maxx = precc(x);
      }
   }
};
```

3.33 Lowest Commom Ancestor (LCA) e queries de caminhos na árvore

$P[i][j]$ = o 2^j-ésimo pai do i-ésimo nó. $D[i][j]$ = distância para o 2^j-ésimo pai do i-ésimo nó. $computeP(root)$ computa as matrizes P e D em $O(n\log n)$. $LCA(u, v)$ retorna um par (LCA, distância) dos nós u e v em $O(\log n)$. CUIDADO: ele usa o tamanho da árvore N e adota indexação em 1!

```
const int neutral = 0;
#define comp(a, b) ((a)+(b))

typedef pair<int, int> ii;
vector<ii> adjList[MAXN];
int level[MAXN], N;
int P[MAXN][MAXLOGN], D[MAXN][MAXLOGN];

void depthdfs(int u) {
   for(int i=0; i<(int)adjList[u].size(); i++) {
      int v = adjList[u][i].first;
      int w = adjList[u][i].second;
      if (v == P[u][0]) continue;
      P[v][0] = u; D[v][0] = w;
      level[v] = 1 + level[u];
      depthdfs(v);
   }
}

void computeP(int root) {
   level[root] = 0;
   P[root][0] = root; D[root][0] = neutral;
```

```
    depthdfs(root);
    for(int j = 1; j < MAXLOGN; j++)
        for(int i = 1; i <= N; i++) {
            P[i][j] = P[P[i][j-1]][j-1];
            D[i][j] = comp(D[P[i][j-1]][j-1], D[i][j
                -1]);
        }
}

ii LCA(int u, int v) {
    if (level[u] > level[v]) swap(u, v);
    int d = level[v] - level[u];
    int ans = neutral;
    for(int i = 0; i < MAXLOGN; i++) {
        if (d & (1<<i)) {
            ans = comp(ans, D[v][i]);
            v = P[v][i];
        }
    }
    if (u == v) return ii(u, ans);
    for(int i = MAXLOGN-1; i >= 0; i--)
        while(P[u][i] != P[v][i]) {
            ans = comp(ans, D[v][i]);
            ans = comp(ans, D[u][i]);
            u = P[u][i]; v = P[v][i];
        }
    ans = comp(ans, D[v][0]);
    ans = comp(ans, D[u][0]);
    return ii(P[u][0], ans);
}
```

Capítulo 4

Paradigmas

4.1 Merge Sort

Algoritmo $O(n \log n)$ para ordenar o vetor em $[a, b]$. *inv* conta o número de inversões do bubble-sort nesse trecho.

```
#define MAXN 100009
long long inv;
int aux[MAXN];

void mergesort(int arr[], int l, int r) {
   if (l == r) return;
   int m = (l + r) / 2;
   mergesort(arr, l, m);
   mergesort(arr, m+1, r);
   int i = l, j = m + 1, k = l;
   while(i <= m && j <= r) {
      if (arr[i] > arr[j]) {
         aux[k++] = arr[j++];
         inv += j - k;
      }
      else aux[k++] = arr[i++];
   }
   while(i <= m) aux[k++] = arr[i++];
   for(i = l; i < k; i++) arr[i] = aux[i];
}
```

4.2 Quick Sort

Algoritmo *Expected* $O(n \log n)$ para ordenar o vetor em $[a, b]$. É o mais rápido conhecido.

```
void quicksort(int* arr, int l, int r) {
    if (l >= r) return;
    int mid = l + (r - l) / 2;
    int pivot = arr[mid];
    swap(arr[mid],arr[l]);
    int i = l + 1, j = r;
    while (i <= j) {
        while(i <= j && arr[i] <= pivot) i++;
        while(i <= j && arr[j] > pivot) j--;
        if (i < j) swap(arr[i], arr[j]);
    }
    swap(arr[i-1], arr[l]);
    quicksort(arr, l, i-2);
    quicksort(arr, i, r);
}
```

4.3 Longest Increasing Subsequence (LIS)

$O(n \log n)$. Ao final de cada iteração i, o k-ésimo elemento (1-indexed) de s é o menor elemento que tem uma subsequência crescente de tamanho k terminando nele.

```
int lis(int arr[], int n) {
   multiset<int> s;
   multiset<int>::iterator it;
   for(int i = 0; i < n; i++) {
      s.insert(arr[i]);
      it = s.upper_bound(arr[i]); //non-decreasing
      //it = ++s.lower_bound(arr[i]); //strictly
          increasing
      if (it != s.end()) s.erase(it);
   }
   return s.size();
}
```

4.4 Problema dos Pares mais Próximos

Implementação $O(n \log n)$ para achar os pares mais próximos segundo a distância euclidiana em uma array de pontos 2D. A implementação original é $O(n \log^2 n)$, mas para muitos pontos, é necessário otimizar com merge sort. Caso precise mudar para pontos inteiros, mudar *dist* para usar quadrado da distância e não esquecer de usar $1+\sqrt{d}$ em vez de d.

```
#define MAXN 100309

struct point {
    double x, y;
    point() { x = y = 0; }
    point(double _x, double _y) : x(_x), y(_y) {}
};

typedef pair<point, point> pp;

double dist(pp p) {
   double dx = p.first.x - p.second.x;
   double dy = p.first.y - p.second.y;
   return hypot(dx, dy);
}

point strip[MAXN];

pp closest(point *P, int l, int r) {
   if (r == l) return pp(point(INF, 0), point(-INF,
       0));
   int m = (l + r) / 2, s1 = 0, s2;
   int midx = (P[m].x + P[m+1].x)/2;
   pp pl = closest(P, l, m);
   pp pr = closest(P, m+1, r);
   pp ans = dist(pl) > dist(pr) ? pr : pl;
   double d = dist(ans);
   for(int i = l; i <= m; i++) {
      if (midx - P[i].x < d) strip[s1++] = P[i];
   }
   s2 = s1;
   for(int i = m+1; i <= r; i++) {
      if (P[i].x - midx < d) strip[s2++] = P[i];
   }
   for(int j = 0, s = s1; j < s1; j++) {
      point p = strip[j];
      for(int i = s; i < s2; i++) {
         point q = strip[i];
         pp cur = pp(p, q);
         double dcur = dist(cur);
         if (d > dcur) {
            ans = cur; d = dcur;
         }
         if (q.y - p.y > d) break;
         if (p.y - q.y > d) s = i+1;
      }
   }
   int i = l, j = m+1, k = l;
   while(i <= m && j <= r) {
      if (P[i].y < P[j].y) strip[k++] = P[i++];
      else strip[k++] = P[j++];
   }
   while(i <= m) strip[k++] = P[i++];
   for(i = l; i < k; i++) P[i] = strip[i];
   return ans;
}

bool compx(point a, point b) {
    return a.x < b.x;
}

pp closest(point *P, int n){
   sort(P, P+n, compx);
   return closest(P, 0, n-1);
}
```

4.5 Otimização de Dois Ponteiros

Reduz a complexidade de $O(n^2k)$ para $O(nk)$ de PD's da seguinte forma (e outras variantes):

$$dp[i][j] = 1 + \min_{1 \le k \le i} (\max(dp[k-1][j-1], dp[i-k][j])),\ caso\ base:\ dp[0][j], dp[i][0] \tag{4.1}$$

- $A[i][j] = k$ ótimo que minimiza $dp[i][j]$.
- É necessário que $dp[i][j]$ seja crescente em i: $dp[i][j] \le dp[i+1][j]$.
- Este exemplo é o problema dos ovos e dos prédios.

```
#include <algorithm>
using namespace std;
#define MAXN 1009
#define MAXK 19
#define INF (1<<30)

int dp[MAXN][MAXK], A[MAXN][MAXK], N, K;

void twopointer() {
   for(int i=0; i<=N; i++) dp[i][0] = INF;
   for(int j=0; j<=K; j++) dp[0][j] = 0, A[0][j] =
       1;
   dp[0][0] = 0;
   for(int i=1; i<=N; i++) {
      for(int j=1; j<=K; j++) {
         dp[i][j] = INF;
         for(int k=A[i-1][j]; k<=i; k++) {
            int cur = 1 + max(dp[k-1][j-1], dp[i-k][
                j]);
            if (dp[i][j] > cur) {
               dp[i][j] = cur;
               A[i][j] = k;
            }
            if (dp[k-1][j-1] > dp[i-k][j]) break;
         }
      }
   }
}
```

4.6 Otimização de Convex Hull Trick

Reduz a complexidade de $O(n^2k)$ para $O(nk \log n)$ de uma PD da seguinte forma (e outras variantes):

$$dp[i][j] = \min_{0 \le k < i}(A[k] * x[i] + dp[k][j-1]),\ caso\ base:\ dp[0][j], dp[i][0] \tag{4.2}$$

É necessário que A seja decrescente: $A[i] \ge A[i+1]$.

```
#define MAXN 1009
class CHTrick{ ... };

ll x[MAXN], A[MAXN], dp[MAXN][MAXN];
int N, K;

void solve() {
    for(int i=0; i<=N; i++) dp[i][0] = 0;
    CHTrick cht(false);
    for(int j=1; j<=K; j++) {
        dp[0][j] = 0;
        cht.clear();
        for(int i=1; i<=N; i++) {
            cht.push(A[i-1], dp[i-1][j-1]);
            dp[i][j] = cht.query(x[i]);
        }
    }
}
```

4.7 Otimização de Slope Trick

Reduz a complexidade de $O(nS^2)$ para $O(n \log n)$ da seguinte PD, onde $f[i] = \min_j(dp[i][j])$ e $opt[i] = j$ que otimiza $f[i]$:

$$dp[i][j] = \min_{k \le j}(dp[i-1][k] + |a[i] - k|),\ caso\ base:\ dp[0][j] = max(0, a[i] - j) \tag{4.3}$$

```
#define MAXN 3009
int N;
ll a[MAXN], f[MAXN], opt[MAXN];

ll slope() {
    priority_queue<ll> pq;
    opt[0] = a[0]; f[0] = 0;
    pq.push(a[0]);
    for(int i=1; i<N; i++) {
        pq.push(a[i]);
        f[i] = f[i-1] + abs(a[i] - pq.top());
        if (a[i] < pq.top()) {
            pq.pop(); pq.push(a[i]);
        }
        opt[i] = pq.top();
    }
    return f[N-1];
}
```

4.8 Otimização de Divisão e Conquista

Reduz a complexidade de $O(n^2k)$ para $O(nk \log n)$ de PD's das seguintes formas (e outras variantes):

$$dp[i][j] = \min_{0 \le k < i}(dp[k][j-1] + C[k][i]),\ caso\ base:\ dp[0][j], dp[i][0] \tag{4.4}$$

- $C[i][k]$ = custo que só depende de i e de k.
- $A[i][j] = k$ ótimo que minimiza $dp[i][j]$.

É necessário que A seja crescente ao longo de cada coluna: $A[i][j] \le A[i+1][j]$.

```
#define MAXN 1009

int dp[MAXN][MAXN], C[MAXN][MAXN], N, K;

void calculatedp(int min_i, int max_i, int j, int
    min_k, int max_k) {
    if (min_i > max_i) return;
    int i = (min_i + max_i)/2;
    int ans = INF, opt;
    for(int k=min_k; k<=min(max_k, i-1); k++) {
        if (ans > dp[k][j-1] + C[k][i]) {
            opt = k;
            ans = dp[k][j-1] + C[k][i];
        }
    }
    dp[i][j] = ans;
    calculatedp(min_i, i-1, j, min_k, opt);
    calculatedp(i+1, max_i, j, opt, max_k);
}

void solve() {
    for(int i=0; i<=N; i++) dp[i][0] = 0;
    for(int j=0; j<=K; j++) dp[0][j] = 0;
    for(int j=1; j<=K; j++) {
        calculatedp(1, N, j, 0, N-1);
    }
}
```

4.9 Otimização de Knuth

Reduz a complexidade de $O(n^3)$ para $O(n^2)$ de PD's das seguintes formas (e outras variantes):

$$dp[i][j] = C[i][j] + \min_{i<k<j}(dp[i][k] + dp[k][j]),\ caso\ base:\ dp[i][j], j-i<S \tag{4.5}$$

$$dp[i][j] = \min_{i<k<j}(dp[i][k] + C[i][k]),\ caso\ base:\ dp[i][j], j-i<S \tag{4.6}$$

- S é uma constante definida, normalmente 1 (caso base $dp[i][i]$).
- $C[i][j]$ = custo que só depende de i e de j.
- $A[i][j] = k$ ótimo que minimiza $dp[i][j]$.

É necessário que se satisfaçam as seguintes condições:

- Desigualdade quadrangular sobre C: $C[a][c] + C[b][d] \leq C[a][d] + C[b][c]$, $a \leq b \leq c \leq d$.
- Monotonicidade sobre C: $C[b][c] \leq C[a][d]$, $a \leq b \leq c \leq d$.

Ou a seguinte condição:

- A crescente nas linhas e nas colunas: $A[i][j-1] \leq A[i][j] \leq A[i+1][j]$.

```
typedef long long ll;

ll dp[MAXN][MAXN], C[MAXN][MAXN];
int A[MAXN][MAXN], N, S;

void knuth() {
    ll cur;
    for(int s = 0; s < N; s++) {
        for(int i=0, j; i + s<N; i++) {
            j = i + s;
            if (s < S) {    //Caso base
                dp[i][j] = 0;
                A[i][j] = i;
                continue;
            }
            dp[i][j] = INFLL;
            for(int k = A[i][j-1]; k <= A[i+1][j]; k++)
                {
                cur = C[i][j] + dp[i][k] + dp[k][j];
                if (dp[i][j] > cur) {
                    dp[i][j] = cur;
                    A[i][j] = k;
                }
            }
        }
    }
}
```

4.10 Otimização de Lagrange

Reduz a complexidade de $O(nk)$ para $O(n \log k)$ de PD's da forma (e variantes): $dp[i][k]$ = algum custo mínimo para os primeiros i elementos (ou a partir de i) de forma que só podemos fazer uma jogada no máximo k vezes.

Em vez de colocar restrição de quantas vezes ainda podemos jogar na última dimensão, resolvemos a PD: $dp[i]$ = algum custo mínimo para os primeiros i elementos (ou a partir de i) de forma a que não haja restrição de k, mas para cada vez que fazemos uma jogada, temos um custo extra C. Suponha que $f(x) = dp[n][x]$ seja uma função convexa, ou seja, $f(x) - f(x-1) \geq f(x+1) - f(x)$, o que acontece quando, ao adicionarmos uma jogada extra fazemos o melhor movimento possível com ela, e a cada nova jogada adicionada, a escolha ótima otimiza menos. Assim, o algoritmo, ao tentar otimizar $dp[i]$, o número de jogadas x usado otimizará $f(x) + Cx$, que é uma função modal. Assim, podemos fazer busca binária de C de forma a que o x que otimiza $dp[i]$ seja igual a k. Quando maior C, menor será x.

```
int N, K;
ll dp[MAXN];

//Computa dp[i] e retorna o numero otimo de jogadas
int computeDp(ll C) { ... }

ll computeC() {
    ll lo = 0;      //computeDp(lo) > K
    ll hi = INFLL; //computeDp(hi) <= K
     //trocar por for(50) para double
     //NAO USAR while(computeDp(hi) != K)
    while(hi > lo + 1) {
        ll C = (hi + lo) / 2;
        if (computeDp(C) > K) lo = C;
        else hi = C;
    }
    return hi;
}

ll solve() {
    double C = ComputeC();
    computeDp(C);
    return dp[N] - C*K;
}
```

Capítulo 5

Grafos

5.1 DFS Spanning Tree

```
#define MAXN 1009
#define UNVISITED -1
#define EXPLORED -2
#define VISITED -3

int vis[MAXN], parent[MAXN];
vector<int> adjList[MAXN];

void graphCheck(int u) { // DFS for checking graph
    edge properties
  vis[u] = EXPLORED;
  for (int j = 0, v; j < (int)adjList[u].size(); j
      ++) {
    v = adjList[u][j];
    if (vis[v] == UNVISITED) {
      printf("␣Tree␣Edge␣(%d,␣%d)\n", u, v);
      parent[v] = u; // parent of this children
          is me
      graphCheck(v);
    }
    else if (vis[v] == EXPLORED) {
      printf("␣Back␣Edge␣(%d,␣%d)␣(Cycle)\n", u,
          v);
    }
    else if (vis[v] == VISITED)
      printf("␣Forward/Cross␣Edge␣(%d,␣%d)\n", u,
          v);
  }
  vis[u] = VISITED;
}
```

5.2 Pontos de articulação e Pontes

```
#define MAXN 1009
#define UNVISITED -1

int num[MAXN], N, low[MAXN], parent[MAXN], counter,
    rootChildren, articulationVertex[MAXN], root;
vector<int> adjList[MAXN];

void tarjan(int u) {
  low[u] = num[u] = counter++;
  for (int j = 0, v; j < (int)adjList[u].size(); j
      ++) {
    v = adjList[u][j];
    if (num[v] == UNVISITED) {
      parent[v] = u;
      if (u == root) rootChildren++;
      tarjan(v);
      if (low[v] >= num[u]) articulationVertex[u]
          = true;
      if (low[v] > num[u]) printf("␣Edge␣(%d,␣%d)
          ␣is␣a␣bridge\n", u, v);
      low[u] = min(low[u], low[v]);
    }
    else if (v != parent[u])
      low[u] = min(low[u], num[v]);
  }
}

int main() {
  counter = 0;
  memset(&num, UNVISITED, sizeof num);
  memset(&low, 0, sizeof low);
  memset(&parent, 0, sizeof parent);
  memset(&articulationVertex, 0, sizeof
      articulationVertex);
  printf("Bridges:\n");
  for (int i = 0; i < N; i++)
    if (num[i] == UNVISITED) {
      root = i; rootChildren = 0; tarjan(i);
      articulationVertex[root] = (rootChildren >
          1);
    } // special case
  printf("Articulation␣Points:\n");
  for (int i = 0; i < N; i++)
    if (articulationVertex[i])
      printf("␣Vertex␣%d\n", i);
  return 0;
}
```

5.3 Ordenação Topológica

Inicializar vis como false. toposort guarda a ordenação na ordem inversa!

```
#define MAXN 1009

int vis[MAXN];
vector<int> adjList[MAXN];
vector<int> toposort;    //Ordem reversa!

void ts(int u) {
   vis[u] = true;
   for (int j = 0, v; j < (int)adjList[u].size(); j
       ++) {
      v = adjList[u][j];
      if (!vis[v]) ts(v);
   }
   toposort.push_back(u);
}
```

5.4 Componentes Fortemente Conexos: Algoritmo de Kosaraju

```
#define MAXN 100009

vector<int> adjList[MAXN], revAdjList[MAXN], ts;
bool vis[MAXN];
int comp[MAXN], parent = 0, numSCC;

void revdfs(int u) {
    vis[u] = true;
    for(int i = 0, v; i < (int)revAdjList[u].size();
         i++) {
      v = revAdjList[u][i];
       if(!vis[v]) revdfs(v);
    }
    ts.push_back(u);
}

void dfs(int u) {
    vis[u] = true; comp[u] = parent;
    for(int i = 0, v; i < (int)adjList[u].size(); i
        ++) {
       v = adjList[u][i];
       if(!vis[v]) dfs(v);
    }
}

void kosaraju(int n) {
   memset(&vis, false, sizeof vis);
    for(int i = 0; i < n; i++)  {
        if(!vis[i]) revdfs(i);
    }
    memset(&vis, false, sizeof vis);
    numSCC = 0;
    for(int i = n-1; i >= 0; i--) {
        if(!vis[ts[i]]) {
            parent = ts[i];
            dfs(ts[i]);
            numSCC++;
        }
    }
}
```

5.5 2-SAT: Algoritmo de Tarjan

```
#define MAXN 200009
#define UNVISITED -1

int num[MAXN], vis[MAXN], comp[MAXN], low[MAXN];
int counter, numSCC;
stack<int> st;
vector<int> adjList[MAXN];

void dfs(int u) {
   low[u] = num[u] = counter++;
   st.push(u); vis[u] = 1;
   for (int j = 0; j < (int)adjList[u].size(); j++)
       {
      int v = adjList[u][j];
      if (num[v] == UNVISITED) dfs(v);
      if (vis[v]) low[u] = min(low[u], low[v]);
   }
   if (low[u] == num[u]) {
      while (true) {
         int v = st.top(); st.pop();
         vis[v] = 0;
         comp[v] = numSCC;
         if (u == v) break;
      }
      numSCC++;
   }
}

void tarjan(int n) {
   counter = numSCC = 0;
   memset(&num, UNVISITED, sizeof num);
   memset(&vis, 0, sizeof vis);
   memset(&low, 0, sizeof low);
   for (int i = 0; i < n; i++) {
      if (num[i] == UNVISITED)
         dfs(i);
   }
}

bool twosat(bool x[], int n) {
   tarjan(2*n);
   for(int i = 0; i < n; i++) {
      if (comp[2*i] == comp[2*i+1]) return false;
      x[i] = (comp[2*i] > comp[2*i+1]);
   }
   return true;
}

void implies(int u, bool pu, int v, bool pv) {
   adjList[2*u+(pu?1:0)].push_back(2*v+(pv?1:0));
   adjList[2*v+(pv?0:1)].push_back(2*u+(pu?0:1));
}
```

5.6 Caminho mínimo: Algoritmo de Dijkstra

Caminho mínimo entre dois nós. Essa implementação funciona para arestas negativas sem ciclos negativos. $O(V \log V + E)$.

```
#define MAXN 100009
vector<ii> adjList[MAXN];

int dijkstra(int s, int t, int n, int dist[]) {
   for(int i = 1; i <= n; i++) dist[i] = INF;
   set<ii> pq;
   dist[s] = 0;
   pq.insert(ii(0, s));
   while(!pq.empty()) {
      int u = pq.begin()->second;
      pq.erase(pq.begin());
      for(int i=0; i<(int)adjList[u].size(); i++) {
         int v = adjList[u][i].second;
         int w = adjList[u][i].first;
         if (dist[v] > dist[u] + w) {
            pq.erase(ii(dist[v], v));
            dist[v] = dist[u] + w;
            pq.insert(ii(dist[v], v));
         }
      }
   }
   return dist[t];
}
```

5.7 Caminho mínimo: Algoritmo de Floyd-Warshall

Caminho mínimo em $O(V^3)$. Muito rápido de codar, pode calcular caminho mínimo para ir e voltar de cada nó. Testado 2,8s para $N = 1000$. No primeiro for, ao final da iteração k (0-indexed), tem-se o resultado parcial considerando apenas arestas dos $k + 1$ primeiros nós. Caso precise dos resultados parciais em determinada ordem, mudar a ordem dos nós.

```
#define MAXN 409
int adjMat[MAXN][MAXN], N;

void floydwarshall() {
   for (int k = 0; k < N; k++)
      for (int i = 0; i < N; i++)
         for (int j = 0; j < N; j++)
            adjMat[i][j] = min(adjMat[i][j],
                adjMat[i][k] + adjMat[k][j]);
}
```

5.8 Caminho mínimo: Algoritmo de Bellman-Ford

Caminho mínimo em grafos com ciclo negativo. $O(VE)$.

```
#define MAXN 1009
int dist[MAXN], N;
typedef pair<int, int> ii;
vector<ii> adjList[MAXN];

int bellmanford(int s, int t) {
   memset(&dist, 1<<20, sizeof dist);
   dist[s] = 0;
   bool hasNegativeWeightCycle = false;
   for (int i = 0, v, w; i < N; i++) {
      for (int u = 0; u < N; u++) {
         for (int j = 0; j < (int)adjList[u].size();
              j++) {
            v = adjList[u][j].first;
            w = adjList[u][j].second;
            if (i==N-1 && dist[v] > dist[u] + w)
               hasNegativeWeightCycle = true;
            else dist[v] = min(dist[v], dist[u] + w)
                ;
         }
      }
   }
   return dist[t];
}
```

5.9 k-ésimo caminho mínimo

Computa o valor do k-ésimo caminho mínimo em $O(kV \log V + kE)$).

```
#define MAXN 109
vector<ii> adjList[MAXN];

void dijkstra(int s, int k, vector<int> kdist[]) {
   map<ii, int> cnt;
   cnt[ii(0, s)]++;
   while(!cnt.empty()) {
      pair<ii, int> cur = *cnt.begin();
      int u = cur.first.second;
      int wu = cur.first.first;
      int am = cur.second;
      cnt.erase(cnt.begin());
      if (am == 0 || kdist[u].size() >= k) continue;
      for(int i = 0; i < am; i++) {
         if (wu == 0 || kdist[u].size() >= k) break;
         kdist[u].push_back(wu);
      }
      for(int i = 0; i < (int)adjList[u].size(); i
          ++) {
         int v = adjList[u][i].second;
         int w = adjList[u][i].first;
         int &cv = cnt[ii(w + wu, v)];
         cv = min(cv + am, k);
      }
   }
}
```

5.10 Caminho mínimo: Shortest Path Faster Algorithm (SPFA)

Otimização de Bellman-Ford. Pior caso $O(VE)$, caso médio igual a dijkstra.

```
#define MAXN 100009
#define INF 0x3f3f3f3f

typedef pair<int, int> ii;
vector<ii> adjList[MAXN];
int dist[MAXN], vis[MAXN], N, M;
bool inq[MAXN];

int spfa(int s, int t) {
   for(int i=0; i<=N; i++) dist[i] = INF;
   memset(&inq, false, sizeof inq);
   memset(&vis, 0, sizeof vis);
   queue<int> q;
   q.push(s); dist[s] = 0;
   inq[s] = true;
   while (!q.empty()) {
      int u = q.front(); q.pop();
      if (vis[u] > N) return -1;
      inq[u] = false;
      for (int i = 0; i < (int)adjList[u].size(); i
          ++) {
         int v = adjList[u][i].second;
         int w = adjList[u][i].first;
         if (dist[u] + w < dist[v]) {
            dist[v] = dist[u] + w;
            if (!inq[v]) {
               vis[v]++; q.push(v);
               inq[v] = true;
            }
         }
      }
   }
   return dist[t];
}
```

5.11 Árvore Geradora Mínima: Algoritmo de Kruskal

Árovre geradora mínima usando Union-Find em $O(E \log V)$.

```
class UnionFind { ... };

int N, M;
vector< pair<ll, ii> > edgeList; // (weight, two
    vertices) of the edge

ll kruskal() {
   ll cost = 0;
   UnionFind UF(N);
   pair<int, ii> edge;
   sort(edgeList.begin(), edgeList.end());
   for (int i = 0; i < M; i++) {
      edge = edgeList[i];
      if (!UF.isSameSet(edge.second.first, edge.
          second.second)) {
         cost += edge.first;
         UF.unionSet(edge.second.first, edge.second.
             second);
      }
   }
   return cost;
}
```

5.12 Árvore Geradora Mínima: Algoritmo de Prim

Árvore geradora mínima sem union find em $O(E \log E)$.

```
#define MAXN 10009
vector<ii> adjList[MAXN];
int N, M;

ll prim() {
   bool taken[MAXN];
   memset(&taken, false, sizeof taken);
   taken[0] = true;
   priority_queue<ii> pq;
   ii v, front; int u, w; ll cost = 0;
   for (int j = 0; j < (int)adjList[0].size(); j++)
       {
      v = adjList[0][j];
      pq.push(ii(-v.second, -v.first));
   }
   while (!pq.empty()) {
      front = pq.top(); pq.pop();
      u = -front.second; w = -front.first;
      if (!taken[u]) {
         cost += (ll)w; taken[u] = true;
         for (int j = 0; j < (int)adjList[u].size();
              j++) {
            v = adjList[u][j];
            if (!taken[v.first]) pq.push(ii(-v.
                second, -v.first));
         }
      }
   }
   return cost;
}
```

5.13 Fluxo Máximo: Algoritmo de Edmonds-Karp

Fluxo máximo de s a t. Pior caso $O(VE^2)$, caso de grafo bipartido $O(VE)$. Usar *add* (precisa adicionar ida e volta na lista de adjacência).

```
#define MAXN 103000
#define MAXM 900000

int ned, prv[MAXN], first[MAXN];
int cap[MAXM], to[MAXM], nxt[MAXM], dist[MAXN];

void init() {
   memset(first, -1, sizeof first);
   ned = 0;
}
void add(int u, int v, int f) {
    to[ned] = v, cap[ned] = f;
    nxt[ned] = first[u];
    first[u] = ned++;
    to[ned] = u, cap[ned] = 0;
    nxt[ned] = first[v];
    first[v] = ned++;
}
int augment(int v, int minEdge, int s) {
   int e = prv[v];
   if (e == -1) return minEdge;
   int f = augment(to[e^1], min(minEdge, cap[e]), s)
       ;
   cap[e] -= f;
   cap[e^1] += f;
   return f;
}
```

```
bool bfs(int s, int t) {
   int u, v;
   memset(&dist, -1, sizeof dist);
   dist[s] = 0;
   queue<int> q; q.push(s);
   memset(&prv, -1, sizeof prv);
   while (!q.empty()) {
      u = q.front(); q.pop();
      if (u == t) break;
      for(int e = first[u]; e!=-1; e = nxt[e]) {
         v = to[e];
         if (dist[v] < 0 && cap[e] > 0) {
            dist[v] = dist[u] + 1;
            q.push(v);
            prv[v] = e;
         }
      }
   }
   return dist[t] >= 0;
}
int edmondskarp(int s, int t) {
   int result = 0;
   while (bfs(s, t)) {
      result += augment(t, INF, s);
   }
   return result;
}
```

5.14 Fluxo Máximo: Algoritmo de Dinic

Fluxo máximo de s a t. Pior caso $O(V^2E)$, no caso médio é praticamente linear, caso de grafo bipartido $O(\sqrt{V}E)$. Usar *add* (precisa adicionar ida e volta na lista de adjacência).

```
#define MAXN 103000
#define MAXM 900000

int ned, first[MAXN], work[MAXN];
int cap[MAXM], to[MAXM], nxt[MAXM], dist[MAXN];

void init() {
   memset(first, -1, sizeof first);
   ned = 0;
}
void add(int u, int v, int f) {
    to[ned] = v, cap[ned] = f;
    nxt[ned] = first[u];
    first[u] = ned++;
    to[ned] = u, cap[ned] = 0;
    nxt[ned] = first[v];
    first[v] = ned++;
}
int dfs(int u, int f, int s, int t) {
   if (u == t) return f;
   int v, df;
   for(int & e = work[u]; e!=-1; e = nxt[e]) {
       v = to[e];
      if (dist[v] == dist[u] + 1 && cap[e] > 0) {
         df = dfs(v, min(f, cap[e]), s, t);
         if (df > 0) {
            cap[e] -= df;
            cap[e^1] += df;
            return df;
         }
      }
   }
   return 0;
}
```

```
bool bfs(int s, int t) {
   int u, v;
   memset(&dist, -1, sizeof dist);
   dist[s] = 0;
   queue<int> q; q.push(s);
   while (!q.empty()) {
      u = q.front(); q.pop();
      for(int e = first[u]; e!=-1; e = nxt[e]) {
         v = to[e];
         if (dist[v] < 0 && cap[e] > 0) {
            dist[v] = dist[u] + 1;
            q.push(v);
         }
      }
   }
   return dist[t] >= 0;
}
int dinic(int s, int t) {
   int result = 0, f;
   while (bfs(s, t)) {
      memcpy(work, first, sizeof work);
      while (f = dfs(s, INF, s, t)) result += f;
   }
   return result;
}
```

5.15 Fluxo Máximo: Algoritmo Push-Relabel

Calcula o fluxo máximo em $O(VE + V^2\sqrt{E})$.

```
#define MAXN 103009
#define MAXM 900000
#define INF 0x3f3f3f3f

int ned, first[MAXN], exc[MAXN], maxhgt[MAXN];
int cap[MAXM], to[MAXM], nxt[MAXM], hgt[MAXN];

void init() {
   memset(first, -1, sizeof first);
   ned = 0;
}

void add(int u, int v, int f) {
   to[ned] = v, cap[ned] = f;
   nxt[ned] = first[u];
   first[u] = ned++;

   to[ned] = u, cap[ned] = 0;
   nxt[ned] = first[v];
   first[v] = ned++;
}

void push(int e) {
   int u = to[e^1], v = to[e];
   int f = exc[u] < cap[e] ? exc[u] : cap[e];
   cap[e] -= f; cap[e^1] += f;
   exc[u] -= f; exc[v] += f;
}

void relabel(int u) {
   int f = INF;
   for (int e = first[u]; e != -1; e = nxt[e]) {
      if (cap[e] > 0 && hgt[to[e]] < f) f = hgt[to[e
          ]];
   }
   if (f < INF) hgt[u] = f + 1;
}

int maxHeight(int s, int t, int n) {
   int sz = 0;
   for (int u = 1; u <= n; u++) {
      if (u == s || u == t || exc[u] == 0) continue;
      if (sz > 0 && hgt[u] > hgt[maxhgt[0]]) sz = 0;
      if (sz == 0 || hgt[u] == hgt[maxhgt[0]])
         maxhgt[sz++] = u;
   }
   return sz;
}

int pushrelabel(int s, int t, int n) {
   memset(&hgt, 0, sizeof hgt);
   memset(&exc, 0, sizeof exc);
   hgt[s] = n; exc[s] = INF;
   for (int e = first[s]; e != -1; e = nxt[e])
      push(e);
   int sz;
   while ((sz = maxHeight(s, t, n)) > 0) {
      for(int i = 0; i < sz; i++) {
         int u = maxhgt[i], pushed = 0;
         for (int e = first[u]; e != -1 && exc[u]; e
             = nxt[e]) {
            if (cap[e] > 0 && hgt[u] == hgt[to[e]] +
                1)
               push(e), pushed = 1;
         }
         if (!pushed) {
            relabel(u); break;
         }
      }
   }
   return exc[t];
}
```

5.16 Corte Mínimo Global: Algoritmo de Stoer-Wagner

Calcula o corte mínimo global em $O(V^3)$: dado um grafo representado com matriz de adjacência, calcula o custo mínimo para desconectar o grafo. Esta implementação modifica o grafo, logo se precisar dele depois deve-se fazer uma cópia.

```
#define MAXN 509
#define INF 0x3f3f3f3f

int n, adjMatrix[MAXN][MAXN];
vector<int> bestCut;

int mincut() {
    int bestCost = INF;
    vector<int> v[MAXN];
    for (int i=0; i<n; ++i)
        v[i].assign (1, i);
    int w[MAXN], sel;
    bool exist[MAXN], added[MAXN];
    memset (exist, true, sizeof exist);
    for (int phase=0; phase<n-1; ++phase) {
        memset (added, false, sizeof added);
        memset (w, 0, sizeof w);
        for (int j=0, prev; j<n-phase; ++j) {
            sel = -1;
            for (int i=0; i<n; ++i) {
                if (exist[i] && !added[i] && (sel ==
                    -1 || w[i] > w[sel]))
                    sel = i;
            }
            if (j == n-phase-1) {
                if (w[sel] < bestCost) {
                    bestCost = w[sel];
                    bestCut = v[sel];
                }
                v[prev].insert (v[prev].end(), v[sel
                    ].begin(), v[sel].end());
                for (int i=0; i<n; ++i) adjMatrix[
                    prev][i] = adjMatrix[i][prev] +=
                     adjMatrix[sel][i];
                exist[sel] = false;
            }
            else {
                added[sel] = true;
                for (int i=0; i<n; ++i)  w[i] +=
                    adjMatrix[sel][i];
                prev = sel;
            }
        }
    }
    return bestCost;
}
```

5.17 Min Cost Max Flow

$O(VE(V \log V + E))$. Usar função *add* para adiconar as arestas. $maxflow$ é o fluxo desejado, ao final do algoritmo $maxflow$ retorna quanto de fluxo não foi possível repassar.

```
#define MAXN 100009
#define MAXM 900009
#define INFLL 0x3f3f3f3f3f3f3f3f

int ned;
int first[MAXN], to[MAXM], nxt[MAXM], prv[MAXN];
ll cap[MAXM], cost[MAXM], dist[MAXN], pot[MAXN];

void init() {
   memset(first, -1, sizeof first);
   ned = 0;
}

void add(int u, int v, ll f, ll c) {
   to[ned] = v, cap[ned] = f;
   cost[ned] = c, nxt[ned] = first[u];
   first[u] = ned++;
   to[ned] = u, cap[ned] = 0;
   cost[ned] = -c, nxt[ned] = first[v];
   first[v] = ned++;
}

bool dijkstra(int s, int t, int n) {
   memset(&prv, -1, sizeof prv);
   for(int i = 1; i <= n; i++) dist[i] = INFLL;
   set< pair<ll, int> > q;
   q.insert(make_pair(0LL, s));
   dist[s] = prv[s] = 0;
   while(!q.empty()) {
      int u = q.begin()->second;
      q.erase(q.begin());
      for(int e = first[u]; e != -1; e = nxt[e]) {
         if (cap[e] <= 0) continue;
         int v = to[e];
         ll new_dist = dist[u] + cost[e] + pot[u] -
             pot[v];
         if (new_dist < dist[v]) {
            q.erase(make_pair(dist[v], v));
            dist[v] = new_dist;
            prv[v] = e;
            q.insert(make_pair(new_dist, v));
         }
      }
   }
   return prv[t] != -1;
}

ll augment(int s, int t, ll &maxflow, int n) {
   ll flow = maxflow;
   for(int i = t; i != s; i = to[prv[i]^1])
      flow = min(flow, cap[prv[i]]);
   for(int i = t; i != s; i = to[prv[i]^1]) {
      cap[prv[i]] -= flow;
      cap[prv[i]^1] += flow;
   }
   maxflow -= flow;
   ll flowCost = flow*(dist[t]-pot[s]+pot[t]);
   for(int i = 1; i <= n; i++)
      if (prv[i] != -1) pot[i] += dist[i];
   return flowCost;
}

ll mincostmaxflow(int s, int t, ll &maxflow, int n){
   ll flowCost = 0;
   memset(pot, 0, sizeof pot);
   while(maxflow > 0 && dijkstra(s, t, n)) {
      flowCost += augment(s, t, maxflow, n);
   }
   return flowCost;
}
```

5.18 Emparelhamento de Custo Mínimo: Algoritmo húngaro

Computa o maximum matching com custo mínimo em um grafo bipartido com pesos nas arestas em $O(n^2m)$. Setar n e m, que são os números de nós em cada lado do grafo bipartido, e preencher a matriz cost[i][j], que representa o custo entre os nós i do lado esquerdo e j do lado direito. Para computar o custo máximo, basta negar a matriz cost[][]. 1-indexed. $pairV[j]$ é o par do elemento j do lado direito.

```
#define MAXN 2009
#define MAXM 2009
#define INF 0x3f3f3f3f

int n, m;
int pu[MAXN], pv[MAXN], cost[MAXN][MAXM];
int pairV[MAXN], way[MAXM], minv[MAXM];
bool used[MAXM];

void hungarian() {
   memset(&pairV, 0, sizeof pairV);
   for(int i = 1, j0 = 0; i <= n; i++) {
      pairV[0] = i;
      memset(&minv, INF, sizeof minv);
      memset(&used, false, sizeof used);
      do {
         used[j0] = true;
         int i0 = pairV[j0], delta = INF, j1;
         for(int j = 1; j <= m; j++) {
            if (used[j]) continue;
            int cur = cost[i0][j] - pu[i0] - pv[j];
            if (cur < minv[j])
               minv[j] = cur, way[j] = j0;
            if (minv[j] < delta)
               delta = minv[j], j1 = j;
         }
         for(int j = 0; j <= m; j++) {
            if (used[j])
               pu[pairV[j]] += delta, pv[j] -= delta
                   ;
            else minv[j] -= delta;
         }
         j0 = j1;
      } while(pairV[j0] != 0);
      do {
         int j1 = way[j0];
         pairV[j0] = pairV[j1];
         j0 = j1;
      } while(j0);
   }
}
```

5.19 Emparelhamento Máximo: Algoritmo de Kuhn

Emparelhamento máximo em grafo bipartido em $O(VE)$. Vértices enumerados de 1 a m em U e de 1 a n em V. Mais rápido de codar do que Hopcroft-Karp.

```
vector<int> adjU[MAXN];
int pairU[MAXN], pairV[MAXN];
bool vis[MAXN];
int m, n;

bool dfs(int u) {
    vis[u] = true;
    if (u == 0) return true;
    for (int i = 0; i < (int)adjU[u].size(); i++) {
        int v = adjU[u][i];
        if (!vis[pairV[v]] && dfs(pairV[v])) {
            pairV[v] = u; pairU[u] = v;
            return true;
        }
    }
    return false;
}
int kuhn() {
    memset(&pairU, 0, sizeof pairU);
    memset(&pairV, 0, sizeof pairV);
    int result = 0;
    for (int u = 1; u <= m; u++) {
        memset(&vis, false, sizeof vis);
        if (pairU[u]==0 && dfs(u)) result++;
    }
    return result;
}
```

5.20 Emparelhamento Máximo: Algoritmo de Hopcroft-Karp

Emparelhamento máximo em grafo bipartido em $O(\sqrt{V}E)$. Vértices enumerados de 1 a m em U e de 1 a n em V.

```
vector<int> adjU[MAXN];
int pairU[MAXN], pairV[MAXN], dist[MAXN];
int m, n;

bool bfs() {
    queue<int> q;
    for (int u = 1; u <= m; u++) {
        if (pairU[u] == 0) {
            dist[u] = 0; q.push(u);
        }
        else dist[u] = INF;
    }
    dist[0] = INF;
    while (!q.empty()) {
        int u = q.front(); q.pop();
        if (dist[u] >= dist[0]) continue;
        for (int i = 0; i < (int)adjU[u].size(); i
            ++) {
            int v = adjU[u][i];
            if (dist[pairV[v]] == INF) {
                dist[pairV[v]] = dist[u] + 1;
                q.push(pairV[v]);
            }
        }
    }
    return (dist[0] != INF);
}
bool dfs(int u) {
    if (u == 0) return true;
    for (int i = 0; i < (int)adjU[u].size(); i++) {
        int v = adjU[u][i];
        if (dist[pairV[v]] == dist[u]+1) {
            if (dfs(pairV[v])) {
                pairV[v] = u; pairU[u] = v;
                return true;
            }
        }
    }
    dist[u] = INF;
    return false;
}
int hopcroftKarp() {
    memset(&pairU, 0, sizeof pairU);
    memset(&pairV, 0, sizeof pairV);
    int result = 0;
    while (bfs()) {
        for (int u=1; u<=m; u++) {
            if (pairU[u]==0 && dfs(u))
                result++;
        }
    }
    return result;
}
```

5.21 Minimum Vertex Cover

Cobertura mínima de um emparelhamento máximo em $O(V + E)$. Vértices enumerados de 1 a m em U e de 1 a n em V.

```
bool Zu[MAXN], Zv[MAXN];
vector<int> coverU, coverV;

void getreach(int u) {
   if (u == 0 || Zu[u]) return;
   Zu[u] = true;
   for (int i = 0; i < (int)adjU[u].size(); i++) {
      int v = adjU[u][i];
      if (v == pairU[u]) continue;
      Zv[v] = true;
      getreach(pairV[v]);
   }
}
void minimumcover() {
   memset(&Zu, false, sizeof Zu);
   memset(&Zv, false, sizeof Zv);
   for (int u = 1; u <= m; u++)
      if (pairU[u] == 0) getreach(u);
   coverU.clear(); coverV.clear();
   for (int u = 1; u <= m; u++)
      if (!Zu[u]) coverU.push_back(u);
   for (int v = 1; v <= n; v++)
      if (Zv[v]) coverV.push_back(v);
}
```

5.22 Maximum Matching: Algoritmo Blossom

Recebe a matriz de adjacência de um grafo qualquer e preenche o vetor *match* com os devidos pares de cada nó. $O(n^3)$.

```
#define MAXN 2005
int mQueue[MAXN], qHead, qTail, match[MAXN],
    cur_blossom[MAXN], parent[MAXN], N;
bool inPath[MAXN], inQueue[MAXN], inBlossom[MAXN];

vector<vector<int> > G;

void push(int u) {
   mQueue[qTail++] = u;
   inQueue[u] = true;
}

int lca(int u, int v) {
   memset(inPath, 0, sizeof(inPath));
   u = cur_blossom[u];
   while(true) {
      inPath[u] = true;
      if(match[u] == -1 || parent[match[u]] == -1)
         break;
      u = cur_blossom[parent[match[u]]];
   }
   v = cur_blossom[v];
   while(true) {
      if(inPath[v]) return v;
      v = cur_blossom[parent[match[v]]];
   }
   return v;
}

int find_aug_path(int R) {
   int u, viz;
   memset(inQueue, 0, sizeof(inQueue));
   memset(parent, -1, sizeof(parent));
   for(int i = 0; i < N; i++) cur_blossom[i] = i;
   qTail = qHead = 0;
   push(R);
   while(qHead < qTail) {
      u = mQueue[qHead++];
      for (int i = 0; i < G[u].size(); i++) {
         viz = G[u][i];
         if(viz != match[u] && cur_blossom[viz] !=
             cur_blossom[u]) {
            if(parent[viz] == -1 && !(match[viz] !=
                -1 && parent[match[viz]] != -1)) {
               if(viz == R) continue;
               parent[viz] = u;
               if(match[viz] == -1) return viz;
               push(match[viz]);
               continue;
            }
            if(viz != R && !(match[viz] != -1 &&
                parent[match[viz]] != -1)) continue;
            int new_blossom = lca(u,viz);
            memset(inBlossom, 0, sizeof(inBlossom));
            int v = u, parent_v = viz;
            while(cur_blossom[v] != new_blossom) {
               inBlossom[cur_blossom[v]] = inBlossom
                   [cur_blossom[match[v]]] = true;
               parent[v] = parent_v;
               parent_v = match[v];
               v = parent[match[v]];
            }
            v = viz, parent_v = u;
            while(cur_blossom[v] != new_blossom) {
               inBlossom[cur_blossom[v]] = inBlossom
                   [cur_blossom[match[v]]] = true;
               parent[v] = parent_v;
               parent_v = match[v];
               v = parent[match[v]];
            }
            for(int i =0; i < N; i++) if(inBlossom[
                cur_blossom[i]]) {
               cur_blossom[i] = new_blossom;
               if(!inQueue[i]) {
                  push(i);
               }
            }
         }
      }
   }
   return -1;
}

void MaxMatch() {
   memset(match, -1, sizeof(match));
   for (int i = 0; i < N; i++) {
      if (match[i] < 0) {
         int v = find_aug_path(i);
         while(v != -1) {
            int parent_v = match[parent[v]];
            match[v] = parent[v];
            match[parent[v]] = v;
            v = parent_v;
         }
      }
   }
}
```

5.23 Gomory Hu Tree: Algoritmo de Gusfield

$V-1$ execuções do algoritmo de fluxo. Computa a Cut Tree de um grafo. A árvore é tal que o min cut entre dois nós (u, v) do grafo é dado pela menor aresta no caminho entre u e v na árvore. Necessário utilizar uma implementação de fluxo, recomendada Edmonds-Karp. *par* é o pai na árvore, *fpar* é o fluxo pro pai. *ghtree* é a lista de adjacência na forma (fluxo, nó). *backup* é o estado original da array *cap* de fluxo. IMPORTANTE: o código usa a variável N e supõe que o grafo está de 1 a N (1-indexed).

```
/* Algoritmo de fluxo antes */

typedef pair<int, int> ii;
int N, par[MAXN], fpar[MAXN]; //parent in tree, flow
    to parent
int backup[MAXM];          //backup for cap
bool inCut[MAXN];
vector<ii> ghtree[MAXN];

void cutGraph(int s) {
   inCut[s] = true;
   for(int e = first[s]; e!=-1; e = nxt[e]) {
      int v = to[e];
      if (!inCut[v] && cap[e] > 0) {
         cutGraph(v);
      }
   }
}

int computeMinCut(int s, int t) {
   memcpy(backup, cap, sizeof cap);
   memset(inCut, false, sizeof inCut);
   int f = edmondskarp(s, t);
   cutGraph(s);
   memcpy(cap, backup, sizeof cap);
   return f;
}

//1-indexed
void gomoryhu() {
   for (int i = 1; i <= N; i++) par[i] = 1;
   for (int s = 2; s <= N; s++) {
      int t = par[s];
      fpar[s] = computeMinCut(s, t);
      for (int u = s+1; u <= N; u++) {
         if (par[u] == t && inCut[u]) {
            par[u] = s;
         }
         if (par[t] == u && inCut[u]) {
            par[s] = u; par[t] = s;
            swap(fpar[s], fpar[t]);
         }
      }
   }
   for (int i = 1; i <= N; i++) ghtree[i].clear();
   for (int i = 1; i <= N; i++) {
      if (par[i] == i) continue;
      ghtree[i].push_back(ii(fpar[i], par[i]));
      ghtree[par[i]].push_back(ii(fpar[i], i));
   }
}
```

5.24 Euler Tour: Algoritmo de Fleury

Computa o Euler Tour em $O(V+E)$. Grafo indexado em 0. Retorna array vazia se não existe caminho. Procura automaticamente o início caso haja algum nó ímpar. Usa a preferência passada pro caso em que todos os nós são ímpares ou 0 se não há preferência.

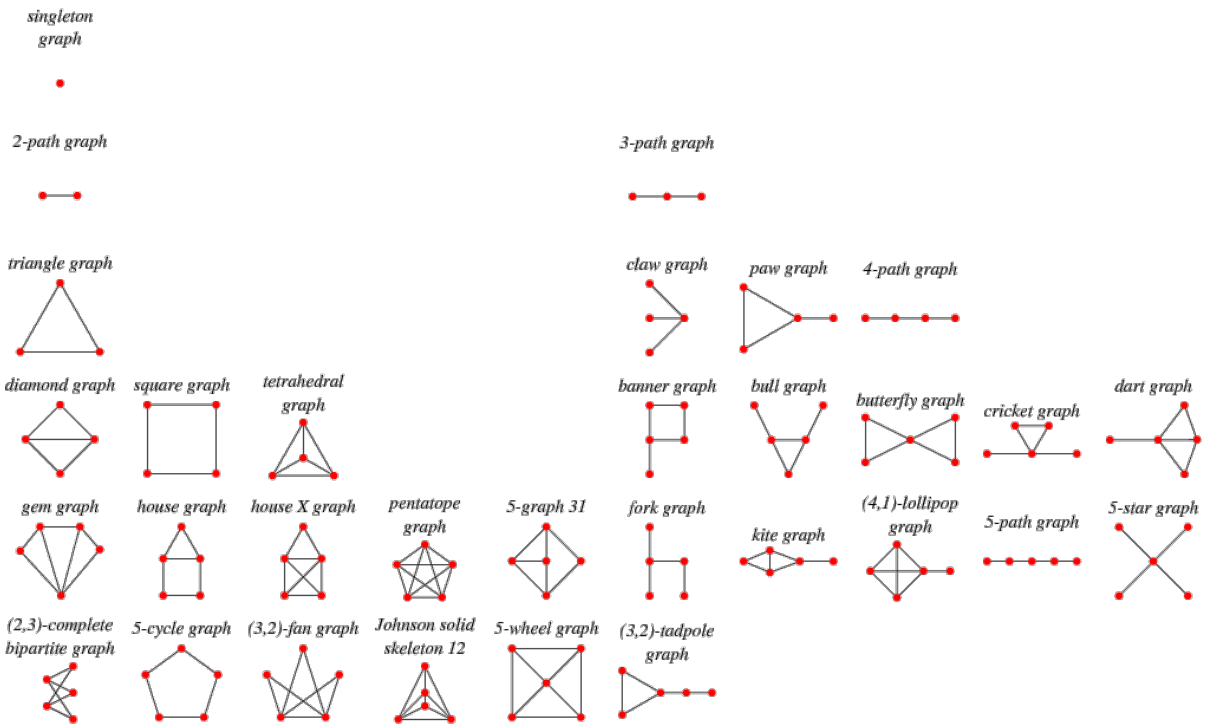
```
#include <vector>
#include <stack>
#define MAXN 400009
using namespace std;

vector<int> adjList[MAXN];
int N, M;

vector<int> euler(int s = 0) {
   vector<int> work(N, 0), in(N, 0), out(N, 0), tour
       ;
   for(int u = 0; u < N; u++) {
      for(int i=0; i<(int)adjList[u].size(); i++) {
         int v = adjList[u][i];
         out[u]++; in[v]++;
      }
   }
   int cntin = 0, cntout = 0;
   for (int u = 0; u < N; u++) {
      if (in[u] == out[u]+1) {
         cntout++;
         if (cntout == 2) return tour;
      }
      else if (out[u] == in[u]+1) {
         cntin++; s = u;
         if (cntin == 2) return tour;
      }
      else if (out[u] != in[u] || in[u]==0)
         return tour;
   }
   stack<int> dfs;
   dfs.push(s);
   while (!dfs.empty()) {
      int u = dfs.top();
      if (work[u] < (int)adjList[u].size()) {
         dfs.push(adjList[u][work[u]++]);
      }
      else {
         tour.push_back(u);
         dfs.pop();
      }
   }
   int n = tour.size();
   for(int i=0; 2*i<n; i++)
      swap(tour[i], tour[n-i-1]);
   return tour;
}
```

5.25 Dominator Tree

Computa a Dominator Tree em $O((V+E)\log V)$. Precisa da lista de adjacência reversa. $sdom[i]$ é o semi-dominator no grafo mapeado por num, $dom[i]$ é o immediate-dominator no grafo mapeado por num. $idom[i]$ é o immediate-dominator real. $dtree$ é a dominator tree. A origem sempre é 1.

```
#include <vector>
using namespace std;
#define MAXN 200009

vector<int> adjList[MAXN], revAdjList[MAXN];
vector<int> dtree[MAXN];
int sdom[MAXN], dom[MAXN], idom[MAXN], N, M;
int dsu[MAXN], best[MAXN]; //auxiliares
int par[MAXN], num[MAXN], rev[MAXN], cnt; //dfs

int find(int x) {
    if (x == dsu[x]) return x;
    int y = find(dsu[x]);
    if (sdom[best[x]] > sdom[best[dsu[x]]]) best[x] =
        best[dsu[x]];
    return dsu[x] = y;
}

void dfs(int u) {
    num[u] = cnt; rev[cnt++] = u;
    for (int i=0; i<(int)adjList[u].size(); i++) {
        int v = adjList[u][i];
        if (num[v] >= 0) continue;
        dfs(v);
        par[num[v]] = num[u];
    }
}

void dominator() {
    for (int u = 1; u <= N; u++) {
        num[u] = -1; dtree[u].clear();
        dsu[u] = best[u] = sdom[u] = u;
    }
    cnt = 0;
    dfs(1);
    for (int j = cnt - 1, u; u = rev[j], j > 0; j--)
        {
        for (int i = 0; i<(int)revAdjList[u].size(); i
            ++) {
            int y = num[revAdjList[u][i]];
            if (y == -1) continue;
            find(y);
            if (sdom[best[y]] < sdom[j]) sdom[j] = sdom
                [best[y]];
        }
        dtree[sdom[j]].push_back(j);
        int x = dsu[j] = par[j];
        for (int i=0; i<(int)dtree[x].size(); i++) {
            int z = dtree[x][i];
            find(z);
            if (sdom[best[z]] < x) dom[z] = best[z];
            else dom[z] = x;
        }
        dtree[x].clear();
    }
    idom[1] = -1;
    for (int i = 1; i < cnt; i++) {
        if (sdom[i] != dom[i]) dom[i] = dom[dom[i]];
        idom[rev[i]] = rev[dom[i]];
        dtree[rev[dom[i]]].push_back(rev[i]);
    }
}
```

5.26 Grafos notáveis



5.27 Teorema de Erdos-Gallai

É condição suficiente para que uma array represente os graus dos vértices de um nó: $d_1 \geq d_2 \geq ... \geq d_n$, $\sum_{i=1}^{n} d_i \equiv 0 \mod 2$, $\sum_{i=1}^{k} d_i \leq k(k-1) + \sum_{i=k+1}^{n} min(d_i, k)$. Para achar esse grafo, usar o algoritmo de Havel-Hakimi: pegar o nó de maior grau restante, conectar ele com os outros nós na ordem descrescente de grau o máximo que der, repetir. Abaixo, implementação $O(n \log n)$ do teorema, otimizável para $O(n)$ se a entrada já estiver ordenada.

```
bool erdosgallai(vector<int> d) {
    sort(d.begin(), d.end(), greater<int>());
    vector<long long> pd(d.size());
    int n = d.size(), p = n-1;
    for(int i = 0; i < n; i++)
        pd[i] = d[i] + (i > 0 ? pd[i-1] : 0);
    for(int k = 1; k <= n; k++) {
        while(p >= 0 && d[p] < k) p--;
        long long sum;
        if (p >= k-1)
            sum = (p-k+1)*1ll*k + pd[n-1] - pd[p];
        else sum = pd[n-1] - pd[k-1];
        if (pd[k-1] > k*(k-1LL) + sum) return false;
    }
    return pd[n-1] % 2 == 0;
}
```

5.28 Teoremas e Fórmulas de Grafos

- **Fórmula de Cayley**: existem n^{n-2} árvores geradoras em um grafo completo de n vértices.
- **Fórmula de Euler para grafos planares**: $V - E + F = 2$, onde F é o número de faces.
- O número de árvores geradores em um grafo bipartido completo é $m^{n-1} \times n^{m-1}$.
- **Teorema de Kirchhoff**: o número de árvores geradoras em um grafo é igual a qualquer cofator da sua matriz laplaciana $L = D - A$, em que D é uma matriz diagonal em que $a_{ii} = d_i$ e A é a matriz de adjacência.
- **Teorema de Konig**: a cobertura mínima de vértices em um grafo bipartido (o número mínimo de vértices a serem removidos para se remover todas as arestas) é igual ao pareamento máximo do grafo. O complementar da cobertura mínima é o conjunto independente máximo - o maior conjunto tal que dois nós não possuem aresta entre si.

  Para computar os vértices na cobertura mínima no grafo bipartido com arestas de L a R, computa-se o emparelhamento máximo e o conjunto U de vértices não-emparelhados em L. O conjunto Z são todos os vértices em U ou alcançavéis a partir de U utilizando arestas alternadas não emparelhado e emparelhado. Os vértices no emparelhamento máximo são $(L - Z) \cup (R \cap Z)$.
- **Teorema de Dilworth**: em um DAG que representa um conjunto parcialmente ordenado, uma cadeia é um subconjunto de vértices tais que todos os pares dentro dele são comparáveis; uma anti-cadeia é um subconjunto tal que todos os pares de vértices dele são não comparáveis. O teorema afirma que a partição mínima não disjunta em cadeias é igual ao comprimento da maior anti-cadeia. Para computar, criar um grafo bipartido: para cada vértice x, duplicar para u_x e v_x. Para cada par comparável x e y com y alcançável por x, adiciona-se $u_x \rightarrow v_y$ (não precisa existir $x \rightarrow y$ no grafo original). O tamanho da partição mínima, também chamada de largura do conjunto, é N menos o emparelhamento máximo. A maior anti-cadeia são os vértices que não estão na cobertura mínima (Teorema de Konig). Caso a partição precise ser disjunta, basta adicionar apenas arestas de elementos imediatamente comparáveis.
- **Teorema de Mirsky**: semelhante ao teorema de Dilworth, o tamanho da partição mínima em anti-cadeias é igual ao comprimento da maior cadeia.
- **Teorema de Menger para arestas**: o número mínimo de arestas que devem ser removidas para separar dois vértices x e y é igual ao número máximo de caminhos com arestas-independentes entre x e y.
- **Teorema de Menger para vértices**: o número mínimo de vértices que devem ser removidos para separar dois vértices x e y não-adjacentes é igual ao número máximo de caminhos com vértices-independentes entre x e y.
- **Código de Prufer**: o código de Prufer de uma árvore pode ser obtido pelo seguinte algoritmo: enquanto a árvore tiver mais de dois nós, pegue a folha de menor índice, remova-a e adicione seu vizinho ao final do código. A relação entre todas as árvores de tamanho n e códigos de tamanho $n-2$ é uma bijeção.
- **Teorema de Hall**: em um grafo bipartido, seja X um subconjunto qualquer dos vértices de um lado e $N_G(X)$ o conjunto de seus vizinhos. Se para todo X, $|X| \leq |N_G(X)|$, então existe um emparelhamento perfeito daquele lado do grafo.
- **Teorema de Dirac**: se o grau de cada vértice de um grafo é pelo menos $n/2$, então existe um caminho hamiltoniano.
- **Teorema de Ore**: se a soma de graus de quaisquer pares de vértices não-adjacentes for pelo menos n, então existe um caminho hamiltoniano.

Capítulo 6

Matemática

6.1 Aritmética Modular

MDC, MMC, euclides extendido, inverso modular $a^{-1}(modm)$, divisão modular $(a/b)(modm)$, exponenciação modular $a^b(modm)$, solução inteira da equação de Diophantine $ax + by = c$. $modMul$ calcula $(a * b)\%m$ sem overflow. Triângulo de Pascal até 10^6. Todos os inversos modulares em relação a um primo p em $O(p)$. Resolve em $O(n \log n)$ o sistema $x \equiv a[i](mod\ p[i])$, $0 \leq i < n$, $gcd(a[i], a[j]) = 1$ para todo $i \neq j$.

```
template <typename T>
T gcd(T a, T b) {
    return b == 0 ? a : gcd(b, a % b);
}

template <typename T>
T lcm(T a, T b) {
    return a * (b / gcd(a, b));
}

template <typename T>
T extGcd(T a, T b, T& x, T& y) {
    if (b == 0) {
        x = 1; y = 0; return a;
    }
    else {
        T g = extGcd(b, a % b, y, x);
        y -= a / b * x; return g;
    }
}

template <typename T>
T modInv(T a, T m) {
    T x, y;
    extGcd(a, m, x, y);
    return (x % m + m) % m;
}

template <typename T>
T modDiv(T a, T b, T m) {
    return ((a % m) * modInv(b, m)) % m;
}

template<typename T>
T modMul(T a, T b, T m) {
   T x = 0, y = a % m;
   while (b > 0) {
      if (b % 2 == 1) x = (x + y) % m;
      y = (y * 2) % m; b /= 2;
   }
   return x % m;
}

template<typename T>
T modExp(T a, T b, T m) {
    if (b == 0) return (T)1;
    T c = modExp(a, b / 2, m);
    c = (c * c) % m;
    if (b % 2 != 0) c = (c*a) % m;
    return c;
}

template<typename T>
void diophantine(T a, T b, T c, T& x, T& y) {
    T d = extGcd(a, b, x, y);
    x *= c / d; y *= c / d;
}

#define MAXN 1000009
typedef long long ll;

ll fat[MAXN];
void preprocessfat(ll m) {
    fat[0] = 1;
    for(ll i=1; i<MAXN; i++)
        fat[i] = (i*fat[i-1])%m;
}

template<typename T>
T pascal(int n, int k, T m) {
    return modDiv(fat[n], (fat[k]*fat[n-k])%m, m);
}

template<typename T>
void allInv(T inv[], T p) {
   inv[1] = 1;
   for (int i = 2; i < p; i++)
      inv[i] = (p - (p/i)*inv[p%i]%p)%p;
}

template<typename T>
T chinesert(T* a, T* p, int n, T m) {
    T P = 1;
    for(int i=0; i<n; i++) P = (P * p[i]) % m;
    T x = 0, pp;
    for(int i=0; i<n; i++) {
        pp = modDiv(P, p[i], m);
        x = (x + (((a[i] * pp) % m) * modInv(pp, p[i
            ]))) % m;
    }
    return x;
}
```

6.2 Teorema Chinês dos Restos generalizado

```
ll crt(ll rem[], ll mod[], int n) {
   if (n == 0) return 0;
    ll ans = rem[0], m = mod[0];
    for (int i = 1; i < n; i++) {
       ll x, y;
       ll g = extGcd(mod[i], m, x, y);
       if ((ans - rem[i])%g != 0) return -1;
       ans = ans + 1ll*(rem[i]-ans)*(m/g)*y;
       m = (mod[i]/g)*(m/g)*g;
       ans = (ans%m + m)%m;
    }
    return ans;
}
```

6.3 Números primos

Diversas operações com números primos. Crivo de Eristótenes, número de divisores, totiente de Euler e número de diferentes fatores primos. *isPrimeSieve* funciona em $O(\sqrt{n}/\log n)$ se os fatores estiverem em *primes*.

```
#define MAXN 10000009
ll sievesize, numDiffPF[MAXN];
bitset<MAXN> bs;
vector<ll> primes;

void sieve(ll n) {
   sievesize = n + 1;
   bs.set();
   bs[0] = bs[1] = 0;
   for (ll i = 2; i <= sievesize; i++) {
      if (bs[i]) {
         for (ll j = i * i; j <= (ll)sievesize; j +=
              i) bs[j] = 0;
         primes.push_back(i);
      }
   }
}
bool isPrimeSieve(ll N) {
   if (N <= (ll)sievesize) return bs[N];
   for (int i = 0; i < (int)primes.size() && primes[
       i]*primes[i] <= N; i++)
      if (N % primes[i] == 0) return false;
   return true;
}
bool isPrime(ll N) {
   if (N < 0) return isPrime(-N);
   for(ll i=2; i*i <= N; i++) {
      if (N % i == 0) return false;
   }
   return true;
}
bool isPrimeFast(ll n) {
    if (n < 0) n = -n;
    if (n < 5 || n % 2 == 0 || n % 3 == 0)
        return (n == 2 || n == 3);
    ll maxP = sqrt(n) + 2;
    for (ll p = 5; p < maxP; p += 6) {
        if (p < n && n % p == 0) return false;
        if (p+2 < n && n % (p+2) == 0) return false;
   }
    return true;
}
```

```
vector<ll> primeFactors(ll N) {
   vector<int> factors;
   ll PF_idx = 0, PF = primes[PF_idx];
   while (PF * PF <= N) {
      while (N % PF == 0) {
         N /= PF;
         factors.push_back(PF);
      }
      PF = primes[++PF_idx];
   }
   if (N != 1) factors.push_back(N);
   return factors;
}
ll numDiv(ll N) {
   ll i = 0, p = primes[i], ans = 1;
   while (p * p <= N) {
      ll power = 0;
      while (N % p == 0) { N /= p; power++; }
      ans *= (power + 1);
      p = primes[++i];
   }
   if (N != 1) ans *= 2;
   return ans;
}
ll eulerPhi(ll N) {
   ll i = 0, p = primes[i], ans = N;
   while (p * p <= N) {
      if (N % p == 0) ans -= ans / p;
      while (N % p == 0) N /= p;
      p = primes[++i];
   }
   if (N != 1) ans -= ans / N;
   return ans;
}
void numDiffPf() {
   memset(numDiffPF, 0, sizeof numDiffPF);
   for (int i = 2; i < MAXN; i++)
      if (numDiffPF[i] == 0)
         for (int j = i; j < MAXN; j += i)
            numDiffPF[j]++;
}
```

6.4 Fórmula de Legendre

Dados um inteiro n e um primo p, calcula o expoente da maior potência de p que divide $n!$ em $O(\log n)$.

```
ll legendre(ll n, ll p) {
   int ans = 0;
   ll prod = p;
   while(prod <= n) {
      ans += n/prod;
      prod *= p;
   }
   return ans;
}
```

6.5 Soma de MDC

Pode-se usar o crivo de Eristótenes para computar o $\phi(x)$ (totiente de Euler) e $F(x)$ para todo x de 1 a n em $O(n \log n)$, ou a fatoração para computar $F(n)$ em $O(\sqrt{n}/\log n)$. F é definida como:

$$F(n) = \sum_{i=1}^{n} gcd(i,n) = \sum_{d|n} d\phi(\frac{n}{d}) = \sum_{d|n} n\frac{\phi(d)}{d} \tag{6.1}$$

```
#define MAXN 200009

typedef unsigned long long ll;
int phi[MAXN], sievesize;
ll f[MAXN];

void gcdsieve(int n) {
   sievesize = n+1;
   for(int i=0; i <= sievesize; i++)
      phi[i] = f[i] = 0;
   for(int i = 1; i <= sievesize; i++) {
      phi[i] += i;
      for(int j = i; j <= sievesize; j += i) {
         if (j > i) phi[j] -= phi[i];
         f[j] += j / i * phi[i];
      }
   }
}

bitset<MAXN> bs;
```

```
vector<ll> primes;
void sieve(ll n) { ... }

ll F(ll N) {
   ll i = 0, p = primes[i], ans = 1;
   while (p * p <= N) {
      if (N % p == 0) {
         int e = 0;
         ll prod = 1;
         while (N % p == 0) {
            N /= p; e++; prod *= p;
         }
         prod /= p;
         ans *= prod * ((p-1)*e + p);
      }
      p = primes[++i];
   }
   ans *= 2*N-1;
   return ans;
}
```

6.6 Crivo linear e funções multiplicativas

Implementação alternativa do crivo de Eristótenes em $O(n)$ e que computa funções multiplicativas: funções f tal que $f(p^k) = g(p,k), p$ primo e $f(pq) = f(p)f(q), gcd(p,q) = 1$. $f(1) = 1$ sempre.

Algumas funções multiplicativas comuns:

- Função constante: $I(p^k) = 1$;
- Função identidade: $Id(p^k) = p^k$;
- Função potência: $Id_a(p^k) = p^{ap}$;
- Função unidade: $\epsilon(p^k) = [k = 1]$;
- Função divisores de grau $a \geq 0$: $\sigma_a(p^k) = \sum_{i=0}^{k} p^{ai}$, $\sigma_a(n) = \sum_{d|n} d^a$;
- Função de Möbius: $\mu(p^k) = [k = 0] - [k = 1]$;
- Função totiente de Euler: $\phi(p^k) = p^k - p^{k-1}$.

```
vector<int> primes;
bitset<MAXN> bs;
int f[MAXN], pw[MAXN];

int g(int p, int k) {
   if (p == 1) return 1;
   return (k==0) - (k==1);
   //int q = 1;
   //for(int i = 1; i < k; i++) q *= p;
   //return q*(p-1);
}
```

```
void sieve(int n) {
   bs.set(); bs[0] = bs[1] = 0;
   primes.clear(); f[1] = g(1, 1);
   for (int i = 2; i <= n; i++) {
      if (bs[i]) {
         primes.push_back(i);
         f[i] = g(i, 1); pw[i] = 1;
      }
      for (int j = 0; j < primes.size() && i*1ll*
           primes[j] <= n; j++) {
         bs[i * primes[j]] = 0;
         if (i % primes[j] == 0) {
            int pwr = 1;
            for(int k = 0; k < pw[i]; k++) pwr *=
                primes[j];
            f[i * primes[j]] = f[i / pwr] * g(primes
                [j], pw[i]+1);
            pw[i * primes[j]] = pw[i] + 1;
            break;
         } else {
            f[i * primes[j]] = f[i] * f[primes[j]];
            pw[i * primes[j]] = 1;
         }
      }
   }
}
```

6.7 Inversão de Möbius

Seja $\mu(n)$ a função de Möbius, cujos valores até n podem ser calculados em $O(n)$ com o crivo linear. Sejam duas funções aritméticas f e g, então a fórmula da inversão de Möbius afirma que:

$$f(n) = \sum_{d|n} g(d) \to g(n) = \sum_{d|n} f(d)\mu(\frac{n}{d}) \tag{6.2}$$

Propriedades importantes:

- $\forall n, 1 = \sum_{d|n} \epsilon(d) \to \epsilon(n) = [n = 1] = \sum_{d|n} \mu(d)$;
- Para funções não aritméticas definidas em $[1, \text{inf})$: $f(x) = \sum_{i=1}^{x} g(\frac{x}{i}) \to g(x) = \sum_{i=1}^{x} \mu(i) f(\frac{x}{i})$;
- Inversão com truncamento: $f(n) = \sum_{i=1}^{n} g(\lfloor \frac{n}{i} \rfloor) \to g(n) = \sum_{i=1}^{n} \mu(i) f(\lfloor \frac{n}{i} \rfloor)$;
- Inversão multiplicativa: $f(n) = \prod_{d|n} g(d) \to g(n) = \prod_{d|n} f(d)^{\mu(\frac{n}{d})}$;
- Número de pares coprimos em $[1, n]$: $\sum_{i=1}^{n} \sum_{j=1}^{n} [gcd(i, j) = 1] = \sum_{i=1}^{n} \sum_{j=1}^{n} \sum_{k|gcd(i,j)} \mu(k) =$
  $= \sum_{k=1}^{n} \mu(k) \sum_{i=1}^{n} [k|i] \sum_{j=1}^{n} [k|j] = \sum_{k=1}^{n} \mu(k) \lfloor \frac{n}{k} \rfloor^2$;
- Função de Möbius em conjuntos parcialmente ordenados: $\mu(s, s) = 1, \mu(s, u) = -\sum_{s \le t < u} \mu(s, t)$;
- Inversão de Möbius em conjuntos parcialmente ordenados: $f(t) = \sum_{s \le t} g(s) \to g(t) = \sum_{s \le t} \mu(s, t) f(s)$.

6.8 Números de Catalan

Números de Catalan podem ser computados pelas fórmulas:

$$Cat(0) = 1 \quad Cat(n) = \frac{4n-2}{n+1} Cat(n-1) = \sum_{i=0}^{n-1} Cat(i) Cat(n-1-i) = \frac{1}{n+1} \binom{2n}{n} = \binom{2n}{n} - \binom{2n}{n+1} \tag{6.3}$$

- $Cat(n)$ = número de arvores binárias completas de $n+1$ folhas ou $2n+1$ elementos;
- $Cat(n)$ = número de combinações válidas para n pares de parêntesis;
- $Cat(n)$ = número de formas que o parentesiamento de $n+1$ elementos pode ser feito;
- $Cat(n)$ = número de triangulações de um polígono convexo de $n+2$ lados; e
- $Cat(n)$ = número de caminhos monotônicos discretos para ir de $(0, 0)$ a (n, n).
- Generalização: número de caminhos para ir de $(0, 0)$ a (x, y) que não cruzam $y - x \ge T = \binom{x+y}{y} - \binom{x+y}{y-T}$.

6.9 Números de Stirling de primeira espécie

Números de Stirling de primeira espécie $s(n, m), n \ge m$ podem ser calculados pela recursão $s(n, m) = s(n-1, m-1) - (n-1)s(n-1, m)$, com $s(n, n) = 1$ e $s(n, 0) = 0, n > 0$

- $s(n, m)$ = coeficiente de x^m em $P(x) = x(x+1)\cdots(x+n-1)$. Usar FFT pra computar $s(n, k), 0 \le k \le n$ em $O(n \log^2 n)$. Ver código abaixo (m = índice do módulo na NTT).
- $|s(n, m)|$ = número de permutações de tamanho n com exatamente m ciclos ou número de formas de alocar n pessoas em m mesas circulares.

```
void compute(vector<ll> & res, int l, int r, int m)
    {
   if (l == r) {
      res = vector<ll>({r, 1LL});
      return;
   }
   vector<ll> a, b;
   compute(a, l, (l + r) / 2, m);
   compute(b, (l + r) / 2 + 1, r, m);
   convolution(a, b, res, m);
}
```

```
void stirling_first(vector<ll> & res, int n, int m)
    {
   if (n == 0) {
      res = vector<ll>({1LL});
      return;
   }
   compute(res, 0, n-1, m);
   while(res.back() == 0) res.pop_back();
}
```

6.10 Números de Stirling de segunda espécie

Números de Stirling de segunda espécie $S(n,m), n \geq m$ podem ser calculados pela recursão: $S(n,m) = S(n-1,m-1) + mS(n-1,m)$, com $S(n,n) = 1$ e $S(n,0) = 0, n > 0$, ou pelo princípio da inclusão-exclusão:

$$S(n,m) = \frac{1}{m!}\sum_{i=0}^{m}(-1)^i\binom{m}{i}(n-i)^n \tag{6.4}$$

- $S(n,m)$ = número de formas de alocar n objetos em exatamente m conjuntos não vazios.
- $S(n,m)m!$ = número de funções sobrejetoras de um conjunto de n elementos em um de m elementos.

6.11 Identidades de soma de binômio

Identidade de Vandermonde, identidade da meia de natal (ou taco de hockey) e teorema multinomial.

$$(x+y)^n = \sum_{i=0}^{n}\binom{n}{i}x^i y^{n-i}, \qquad \binom{m+n}{k} = \sum_{i=0}^{k}\binom{m}{i}\binom{n}{k-i}, \qquad \binom{k+n}{k-1} = \sum_{i=0}^{k-1}\binom{n+i}{i} \tag{6.5}$$

$$(x_1 + x_2 + \cdots + x_m)^n = \sum_{k_1+k_2+\cdots+k_m=n}\binom{n}{k_1, k_2, \cdots, k_m}\prod_{t=1}^{m}x_t^{k_t} \tag{6.6}$$

6.12 Lemma de Burnside e Teorema da Enumeração de Pólya

Seja X um conjunto e G um conjunto de transformações de elementos de X em outros elementos de X. Para cada transformação $g \in G$, tem-se $I(g)$ elementos $x \in X$ tal que $g(x) = x$. O subconjunto máximo X/G de X é tal que se $x, y \in X/G$, não existe $g \in G$ tal que $g(x) = y$. Outra forma de se pensar é que cada elemento $x \in X$ é uma *representação* e está associada a um único *objeto*. Um *objeto* pode ter várias *representações* associadas a si. G é o conjunto de transformações transitivas e invariantes, ou seja, para todo $g \in G$, se dois elementos x e y estão associados a um mesmo objeto, então $g(x)$ e $g(y)$ também estão. $|X/G|$ representa o número de classes de equivalências de G, o número de *objetos* distintos. O lemma de Burnside é dado pela fórmula à esquerda. Caso G seja um conjunto de permutações, seja $C(g)$ o número de ciclos de uma permutação g e k o número de possíveis elementos para preencher a array que representa um elemento $x \in X$. O teorema da enumeração de Pólya é dado à direita. CUIDADO: G deve ser transitivo, ou seja, se $f, g \in G$, então $f \circ g \in G$.

$$|X/G| = \frac{1}{|G|}\sum_{g\in G}I(g) \qquad |X/G| = \frac{1}{|G|}\sum_{g\in G}k^{C(g)} \tag{6.7}$$

6.13 Teste de Primalidade de Miller-Rabin

$O(k\log^2 n)$. Probabilístico, mas provado correto para $n < 2^{64}$ com $k = 9$.

```
template<typename T>
T modMulExp(T a, T b, T m) {
    if (b == 0) return (T)1;
    T c = modMulExp(a, b / 2, m);
    c = modMul(c, c, m);
    if (b % 2 != 0) c = modMul(c, a, m);
    return c;
}

bool miller(long long n) {
   const int pn = 9;
   const int p[] = {2, 3, 5, 7, 11, 13, 17, 19, 23};
   for (int i = 0; i < pn; i++)
      if (n % p[i] == 0) return n == p[i];
   if (n < p[pn - 1]) return false;
   long long s = 0, t = n - 1;
   while (~t & 1) t >>= 1, ++s;
   for (int i = 0; i < pn; ++i) {
      long long pt = modMulExp((long long)p[i], t, n
          );
      if (pt == 1) continue;
      bool ok = false;
      for (int j = 0; j < s && !ok; j++) {
         if (pt == n - 1) ok = true;
         pt = modMul(pt, pt, n);
      }
      if (!ok) return false;
   }
   return true;
}
```

6.14 Algoritmo de Pollard-Rho

Retorna um fator de n, usar para $n > 9 \times 10^{13}$.

```
template<typename T>
T pollard(T n) {
    int i = 0, k = 2, d;
    T x = 3, y = 3;
    while (++i) {
        x = (modMul(x, x, n) + n - 1) % n;
        d = gcd(abs(y - x), n);
        if (d != 1 && d != n) return d;
        if (i == k) y = x, k *= 2;
    }
}
```

6.15 Baby-Step Giant-Step para Logaritmo Discreto

Resolve a equação $a^x \equiv b \mod m$ em $O(\sqrt{m} \log m)$. Retorna -1 se não há solução.

```
template <typename T>
T baby (T a, T b, T m) {
    a %= m; b %= m;
    T n = (T) sqrt (m + .0) + 1, an = 1;
    for (T i=0; i<n; ++i) an = (an * a) % m;
    map<T,T> vals;
    for (T i=1, cur=an; i<=n; ++i) {
        if (!vals.count(cur)) vals[cur] = i;
        cur = (cur * an) % m;
    }
    for (T i=0, cur=b; i<=n; ++i) {
        if (vals.count(cur)) {
            T ans = vals[cur] * n - i;
            if (ans < m) return ans;
        }
        cur = (cur * a) % m;
    }
    return -1;
}
```

6.16 Jogo de Nim e teorema de Sprague-Grundy

- Jogo de nim: dois jogadores navegam por um DAG (normalmente modelado a partir das regras de um jogo), em cada jogada um escolhe por qual aresta andar. Cada nó é um estado do jogo. Existe um estado final que é uma posição perdedora (se os jogadores estiverem nele, o jogador atual perde).
- Equivalente de nim g: o jogador atual tem estratégia vencedora se e somente se o equivalente de nim g_u do estado atual u é diferente de zero, ou seja, $g_u \neq 0$ (Teorema de Bouton).
- Teorema de Sprague-Grundy: em um estado u, $g_u = 0$ se for o estado final, $g_u = mex_v(g_v) \forall v$ alcançável por u. $mex(a_1, a_2, ..., a_n)$ é o primeiro número maior que ou igual a zero que não aparece no conjunto $\{a_1, a_2, ..., a_n\}$.
- Jogo de nim em paralelo: se $u_1, u_2, ..., u_n$ forem os estados atuais de cada jogo jogado em paralelo, o equivalente de nim geral é o xor de todos os individuais $(g_{u_1} \oplus g_{u_2} \oplus ... \oplus g_{u_n})$.
- Misère nim: é igual ao jogo de nim, mas o estado final é uma posição vencedora. Caso exista algum estado u com $g_u > 1$, então usa-se o equivalente de nim. Caso contrário, usa-se a paridade do número de estados mais um.
- Método de Steve Halim: é muito comum usar o método para imprimir todos os possíveis $mex[i]$ no código, pois o grafo normalmente é descrito pelas regras de um jogo e é, portanto, constante.

```
#include <set>
using namespace std;
#define MAXN 10009

void stevehalim() {
    for(int n=1; n<MAXN; n++) {
        if (n <= 2) { mex[n] = 0; continue; }
        set<int> jog;
        for(int i=3; i<=n-2; i++) {
            jog.insert(mex[i-1]^mex[n-i]);
        }
        int cnt = 0;
        while(jog.count(cnt)) cnt++;
        mex[n] = cnt;
    }
    //for(int n=1; n<MAXN; n++)
    //    printf("%d,", mex[n]);
}
```

6.17 Triplas Pitagóricas

Todas as triplas pitagóricas (a, b, c), $a^2 + b^2 = c^2$ podem ser geradas a partir das equações ($k = 1$ gera triplas primitivas):

$$a = k(m^2 - n^2), \quad b = 2kmn, \quad c = k(m^2 + n^2) \tag{6.8}$$

6.18 Matrizes

```
typedef vector< vector< double > > matrix;

matrix operator +(matrix a, matrix b) {
   int n = (int)a.size();
   int m = (int)a[0].size();
   matrix c;
   c.resize(n);
   for(int i=0; i<n; i++) {
      c[i].resize(m);
      for(int j=0; j<m; j++)
         c[i][j] = a[i][j] + b[i][j];
   }
   return c;
}

matrix operator *(matrix a, matrix b) {
   int n = (int)a.size();
   int m = (int)b.size();
   int p = (int)b[0].size();
   matrix c(n, vector<double>(p));
   vector<double> col(m);
   for (int j = 0; j < p; j++) {
      for (int k = 0; k < m; k++)
         col[k] = b[k][j]; //cache friendly
      for (int i = 0; i < n; i++) {
         double s = 0;
         for (int k = 0; k < m; k++)
            s += a[i][k] * col[k];
         c[i][j] = s;
      }
   }
   return c;
}

matrix operator *(double k, matrix a) {
   int n = (int)a.size();
   int m = (int)a[0].size();
   for(int i=0; i<n; i++) for(int j=0; j<m; j++)
      a[i][j] *= k;
   return a;
}

matrix operator -(matrix a, matrix b) {
   return a + ((-1.0) * b);
}
```

6.19 Exponenciação de matrizes e Fibonacci

Calcula o n-ésimo termo de fibonacci em tempo $O(\log n)$. Calcula uma matriz $m \times m$ elevado a n em $O(m^3 \log n)$.

```
matrix matrixExp(matrix a, int n) {
   if (n == 0) return id(a.size());
   matrix c = matrixExp(a, n/2);
   c = c*c;
   if (n%2 != 0) c = c*a;
   return c;
}

matrix fibo() {
   matrix c(2, vector<double>(2, 1));
   c[1][1] = 0;
   return c;
}

double fibo(int n) {
   matrix f = matrixExp(fibo(), n);
   return f[0][1];
}
```

6.20 Sistemas Lineares: Determinante e Eliminação de Gauss

$gauss(A, B)$ retorna se o sistema $Ax = B$ possui solução e executa a eliminação de Gauss em A e B. $det(A)$ computa o determinante em $O(n^3)$ por Eliminação de Gauss.

```
void switchLines(matrix & a, int i, int j) {
   int m = (int)a[i].size();
   for(int k = 0; k < m; k++)
      swap(a[i][k], a[j][k]);
}

void lineSumTo(matrix & a, int i, int j, double c) {
   int m = (int)a[0].size();
   for(int k = 0; k < m; k++) a[j][k] += c*a[i][k];
}

bool gauss(matrix & a, matrix & b, int & switches) {
   switches = 0;
   int n = (int)a.size();
   int m = (int)a[0].size();
   for(int i = 0, l; i < min(n, m); i++) {
      l = i;
      while(l < n && fabs(a[l][i]) < EPS) l++;
      if (l == n) return false;
      switchLines(a, i, l);
      switchLines(b, i, l);
      switches++;
      for(int j=0; j<n; j++) {
         if (i == j) continue;
         double p = -a[j][i] / a[i][i];
         lineSumTo(a, i, j, p);
         lineSumTo(b, i, j, p);
      }
   }
   return true;
}

double det(matrix a) {
   int n = a.size();
   matrix b(n);
   for(int i=0; i<n; i++) b[i].resize(1);
   int sw = 0;
   if (gauss(a, b, sw)) {
      double ans = 1;
      for(int i=0; i<n; i++) ans *= a[i][i];
      return sw % 2 == 0 ? ans : -ans;
   }
   return 0.0;
}
```

6.21 Multiplicação de matriz esparsa

Multiplica duas matrizes em $O(n^2m)$, onde m é o mínimo do número médio de números não nulos em cada linha e coluna.

```
vector< vector<int> > adjA, adjB;

matrix sparsemult(matrix a, matrix b) {
   int n = (int)a.size();
   //assert(a[0].size() == b.size());
   int m = (int)b.size();
   int p = (int)b[0].size();
   adjA.resize(n);
   for(int i=0; i<n; i++) {
      adjA[i].clear();
      for(int k=0; k<m; k++)
         if (fabs(a[i][k]) > EPS)
            adjA[i].push_back(k);
   }
   adjB.resize(p);
   for(int j=0; j<p; j++) {
      adjB[j].clear();
      for(int k=0; k<m; k++)
         if (fabs(b[k][j]) > EPS)
            adjB[j].push_back(k);
   }
   matrix c;
   c.resize(n);
   for(int i=0; i<n; i++) {
      c[i].assign(p, 0);
      for(int j=0; j<p; j++)
         for(int u=0, v=0, k; u<(int)adjA[i].size()
             && v<(int)adjB[j].size();) {
            if (adjA[i][u] > adjB[j][v]) v++;
            else if (adjA[i][u] < adjB[j][v]) u++;
            else {
               k = adjA[i][u];
               c[i][j] += a[i][k]*b[k][j];
               u++; v++;
            }
         }
   }
   return c;
}
```

6.22 Método de Gauss-Seidel

Resolve o sistema iterativamente em $O(n^2 \log PREC^{-1})$. É necessário que a diagonal principal seja dominante.

```
matrix gaussSeidel(matrix & a, matrix & b, double
    PREC) {
   int n = (int)a.size();
   matrix x = b, xp = b;
   double error;
   do {
      error = 0.0;
      for(int i=0; i<n; i++) {
         xp[i][0] = b[i][0];
         for(int j=0; j<n; j++) {
            if (i < j) xp[i][0] -= a[i][j]*xp[j][0];
            if (i > j) xp[i][0] -= a[i][j]*x[j][0];
         }
         xp[i][0] /= a[i][i];
         error = max(error, fabs(xp[i][0]-x[i][0]));
      }
      x = xp;
   } while(error > PREC);
   return xp;
}
```

6.23 XOR-SAT

Executa a eliminação gaussiana com xor sobre o sistema $Ax = b$ em $O(nm^2/64)$.

```
#define MAXN 2009

vector<int> B;
vector< bitset<MAXN> > A;
bitset<MAXN> x;

bool check() {
   int n = A.size(), m = MAXN;
   for(int i = 0; i < n; i++) {
      int acum = 0;
      for(int j = 0; j < m; j++) {
         if (A[i][j]) acum ^= x[j];
      }
      if (acum != B[i]) return false;
   }
   return true;
}
bool gaussxor() {
   int cnt = 0, n = A.size(), m = MAXN;
   bitset<MAXN> vis; vis.set();
   for(int j = m-1, i; j >= 0; j--) {
      for(i = cnt; i < n; i++) {
         if (A[i][j]) break;
      }
      if (i == n) continue;
      swap(A[i], A[cnt]); swap(B[i], B[cnt]);
      i = cnt++; vis[j] = 0;
      for(int k = 0; k < n; k++) {
         if (i == k || !A[k][j]) continue;
         A[k] ^= A[i]; B[k] ^= B[i];
      }
   }
   x = vis;
   for(int i = 0; i < n; i++) {
      int acum = 0;
      for(int j = 0; j < m; j++) {
         if (!A[i][j]) continue;
         if (!vis[j]) {
            vis[j] = 1;
            x[j] = acum^B[i];
         }
         acum ^= x[j];
      }
      if (acum != B[i]) return false;
   }
   return true;
}
```

6.24 Fast Fourier Transform (FFT)

Usar em caso de double. Em caso de inteiro converter com $int(a[i].real() + 0.5)$. Usar struct caso precise ser rápido.

```
#include <complex>
using namespace std;

typedef complex<double> base;

void fft(vector<base> &a, bool invert) {
    int n = (int)a.size();
    for(int i = 1, j = 0; i < n; i++) {
        int bit = n >> 1;
        for(; j >= bit; bit >>= 1) j -= bit;
        j += bit;
        if (i < j) swap(a[i], a[j]);
    }
    for(int len = 2; len <= n; len <<= 1) {
        double ang = 2*acos(-1.0)/len * (invert ? -1 :
            1);
        base wlen(cos(ang), sin(ang));
        for(int i = 0; i < n; i += len) {
            base w(1);
            for(int j = 0; j < len/2; j++) {
                base u = a[i+j], v = a[i+j+len/2] * w;
                a[i + j] = u + v;
                a[i + j + len/2] = u - v;
                w *= wlen;
            }
        }
    }
    for (int i = 0; invert && i < n; i++) a[i] /= n;
}

void convolution(vector<base> a, vector<base> b,
     vector<base> &res) {
    int n = 1;
    while(n < max(a.size(), b.size())) n <<= 1;
    n <<= 1;
    a.resize(n), b.resize(n);
    fft(a, false); fft(b, false);
    res.resize(n);
    for(int i=0; i<n; ++i) res[i] = a[i]*b[i];
    fft(res, true);
}
```

6.25 Number Theoretic Transform (NTT)

Usar long long. Cuidado com overflow. m é o primo selecionado. O resultado é calculado $mod[m]$.

```
template <typename T>
T extGcd(T a, T b, T& x, T& y) { ... }

template <typename T>
T modInv(T a, T m) { ... }

const ll mod[3] = {1004535809LL, 1092616193LL,
    998244353LL};
const ll root[3] = {12289LL, 23747LL, 15311432LL};
const ll root_1[3] = {313564925LL, 642907570LL,
    469870224LL};
const ll root_pw[3] = {1LL<<21, 1LL<<21, 1LL<<23};

void ntt(vector<ll> & a, bool invert, int m) {
    ll n = (ll)a.size();
    for(ll i = 1, j = 0; i < n; i++) {
        ll bit = n >> 1;
        for (; j >= bit; bit >>= 1) j -= bit;
        j += bit;
        if (i < j) swap(a[i], a[j]);
    }
    for(ll len = 2, wlen; len <= n; len <<= 1) {
        wlen = invert ? root_1[m] : root[m];
        for (ll i = len; i < root_pw[m]; i <<= 1)
            wlen = (wlen * wlen % mod[m]);
        for(ll i = 0; i < n; i += len) {
            for(ll j = 0, w = 1; j < len/2; j++) {
                ll u = a[i+j], v = a[i+j+len/2] * w %
                    mod[m];
                a[i+j] = (u+v < mod[m] ? u+v : u+v-mod[m
                    ]);
                a[i+j+len/2] = (u-v >= 0 ? u-v : u-v+mod
                    [m]);
                w = w * wlen % mod[m];
            }
        }
    }
    if (invert) {
        ll nrev = modInv(n, mod[m]);
        for (ll i=0; i<n; ++i)
            a[i] = a[i] * nrev % mod[m];
    }
}

void convolution(vector<ll> a, vector<ll> b, vector<
    ll> & res, int m) {
    ll n = 1;
    while(n < max (a.size(), b.size())) n <<= 1;
    n <<= 1;
    a.resize(n), b.resize(n);
    ntt(a, false, m); ntt(b, false, m);
    res.resize(n);
    for(int i=0; i<n; ++i)
        res[i] = (a[i]*b[i])%mod[m];
    ntt(res, true, m);
}
```

6.26 Convolução circular

Utiliza FFT/NTT para computar em $O(n \log n)$: $res[i] = \sum_{j=0}^{n-1} a[j]b[(i-j+n)\%n]$

```
void circularConv(vector<base> &a, vector<base> &b,
     vector<base> &res) { // fft
    //assert(a.size() == b.size());
    int n = a.size();
    convolution(a, b, res);
    for(int i = n; i < (int)res.size(); i++)
        res[i%n] += res[i];
    res.resize(n);
}
```

6.27 Números complexos

```
struct base { // faster than complex<double>
   double x, y;
   base() : x(0), y(0) {}
   base(double a, double b=0) : x(a), y(b) {}
   base operator/=(double k) { x/=k; y/=k; return (*
       this); }
   base operator*(base a) const { return base(x*a.x
       - y*a.y, x*a.y + y*a.x); }
   base operator*=(base a) {
      double tx = x*a.x - y*a.y;
      double ty = x*a.y + y*a.x;
      x = tx; y = ty;
      return (*this);
   }
   base operator=(double a) { x=a; y=0; return (*
       this); }
   base operator+(base a) const { return base(x+a.x,
        y+a.y); }
   base operator+=(base a) { x+=a.x; y+=a.y; return
       (*this); }
   base operator-(base a) const { return base(x-a.x,
        y-a.y); }
   base operator-=(base a) { x-=a.x; y-=a.y; return
       (*this); }
   double& real() { return x; }
   double& imag() { return y; }
};
```

6.28 Divisão de polinômios

Divisão e resto polinômios em $O(n \log n)$, em que n é o grau do dividendo. $inverse(A(x), t)$ computa os t primeiros termos da série infinita $1/A(x)$ em $O(n \log n)$.

```
void inverse(vector<base> &a, int t) {
   if (t == 1) {
      a.resize(1); a[0].real() = 1.0/a[0].real();
      return;
   }
   vector<base> a0 = a;
   inverse(a0, (t/2) + (t%2));
   convolution(a0, a, a);
   a0.resize(t), a.resize(t);
   a[0] -= base(1.0);
   convolution(a0, a, a);
   a0.resize(t), a.resize(t);
   for(int i = 0; i < t; i++) a[i] = a0[i] - a[i];
}
void divide(vector<base> &a, vector<base> &b, vector
    <base> &d) {
   int n = a.size(), m = b.size();
   if (n < m) {
      d.clear(); d.push_back(0);
      return;
   }
   vector<base> ar = a, br = b;
   reverse(ar.begin(), ar.end());
   reverse(br.begin(), br.end());
   inverse(br, n - m + 1);
   convolution(ar, br, d);
   d.resize(n - m + 1);
   reverse(d.begin(), d.end());
}
void remainder(vector<base> &a, vector<base> &b,
    vector<base> &r) {
   int n = a.size(), m = b.size();
   if (n < m) { r = a; return; }
   vector<base> d, aux;
   divide(a, b, d);
   r = a;
   convolution(d, b, aux);
   r.resize(m-1); aux.resize(m-1);
   for(int i = 0; i < m-1; i++) r[i] -= aux[i];
}
```

6.29 Evaluação em múltiplos pontos

$roots(x[], n)$ computa o polinômio $A(x) = \prod_{i=0}^{n-1}(x - x[i])$ em $O(n \log^2 n)$. $multievaluate(A(x), x, y, n)$ calcula $y[i] = A(x[i]) \forall 0 \leq i < n$ em $O(n \log^2 n)$.

```
#define MAXN 10000

vector<base> rt[4*MAXN];
void roots(int u, double x[], int i, int j) {
   if (i == j) {
      rt[u].resize(2); rt[u][0] = -x[i];
      rt[u][1] = 1.0; return;
   }
   int m = (i + j) / 2;
   roots(2*u, x, i, m); roots(2*u+1, x, m+1, j);
   convolution(rt[2*u], rt[2*u+1], rt[u]);
   rt[u].resize(j-i+2);
}
void roots(vector<base> &a, double x[], int n) {
   roots(1, x, 0, n-1); a = rt[1];
}
vector<base> et[4*MAXN];
void multievaluate(int u, double x[], double y[],
    int i, int j) {
   if (i == j) {
      y[i] = 0;
      double p = 1;
      for(int k = 0; k < (int)et[u].size(); k++)
         y[i] += et[u][k].real()*p, p *= x[i];
      return;
   }
   remainder(et[u], rt[2*u], et[2*u]);
   remainder(et[u], rt[2*u+1], et[2*u+1]);
   int m = (i + j) / 2;
   multievaluate(2*u, x, y, i, m);
   multievaluate(2*u+1, x, y, m+1, j);
}
void multievaluate(vector<base> &a, double x[],
    double y[], int n) {
   roots(1, x, 0, n-1); et[1] = a;
   multievaluate(1, x, y, 0, n-1);
}
```

6.30 Interpolação de polinômios

Calcula o polinômio de grau n que passa pelos pontos $(x[i], y[i]), 0 \leq i < n$ em $O(n \log^3 n)$. Também pode-se resolver em $O(n^2 \log n)$ com o polinômio interpolador de Lagrange $p(x) = \sum_{i=0}^{n-1} \prod_{j \not= i} \frac{x-x[j]}{x[i]-x[j]}$ ou em $O(n^3)$ com eliminassão gaussiana.

```
vector<double> y[MAXN];
double ya0[MAXN], yp[MAXN];
void interpolate(vector<base> &a, int u, double x[],
     int n) {
   if (n == 1) {
      a.resize(1); a[0] = y[u][0];
      return;
   }
   y[2*u] = y[u]; y[2*u].resize(n/2);
   vector<base> a0, a1, p;
   interpolate(a0, 2*u, x, n/2);
   roots(p, x, n/2);
   int m = n-(n/2);
   multievaluate(a0, x+(n/2), ya0, m);
   multievaluate(p,  x+(n/2), yp, m);
   y[2*u+1].resize(m);
   for(int i = 0; i < m; i++)
      y[2*u+1][i] = (y[u][n/2 + i] - ya0[i]) / yp[i
          ];
   interpolate(a1, 2*u+1, x+(n/2), m);
   convolution(p, a1, a);
   int sz = max(a.size(), a0.size());
   a.resize(sz);
   for(int i = 0; i < sz && i < n/2; i++)
      a[i] += a0[i];
   a.resize(n);
}
void interpolate(vector<base> &a, double x[], double
     ry[], int n) {
   y[1].resize(n);
   for(int i = 0; i < n; i++) y[1][i] = ry[i];
   interpolate(a, 1, x, n);
}
```

6.31 Fast Walsh–Hadamard Transform

Computa a convolução com XOR: o termo $a[i]b[j]$ é somado em $c[i \oplus j]$, OR: o termo $a[i]b[j]$ é somado em $c[i|j]$, AND: o termo $a[i]b[j]$ é somado em $c[i\&j]$ em $O(n \log n)$. Em caso de inteiro converter com $fa[i] + 0.5$.

```
#include <vector>
using namespace std;

//int mat[2][2] = {{1, 1}, {1, -1}}, inv[2][2] =
    {{1, 1}, {1, -1}}; //xor
int mat[2][2] = {{1, 1}, {1, 0}}, inv[2][2] = {{0,
    1}, {1, -1}}; //or
//int mat[2][2] = {{0, 1}, {1, 1}}, inv[2][2] =
    {{-1, 1}, {1, 0}}; //and

void fwht(vector<double> & a, bool invert) {
   int n = (int)a.size();
   double u, v;
   for (int len = 1; 2 * len <= n; len <<= 1) {
      for (int i = 0; i < n; i += 2 * len) {
         for (int j = 0; j < len; j++) {
            u = a[i + j];
            v = a[i + len + j];
            if (!invert) {
               a[i + j] = mat[0][0]*u + mat[0][1]*v;
               a[i + len + j] = mat[1][0]*u + mat
                   [1][1]*v;
            }
            else {
               a[i + j] = inv[0][0]*u + inv[0][1]*v;
               a[i + len + j] = inv[1][0]*u + inv
                   [1][1]*v;
            }
         }
      }
   }
   //for (int i=0; invert && i<n; ++i) a[i] /= n; //
       xor
}

void convolution(vector<double> a, vector<double> b,
     vector<double> & res) {
   int n = 1;
   while(n < max(a.size(), b.size())) n <<= 1;
   a.resize(n), b.resize(n);
   fwht(a, false); fwht(b, false);
   res.resize(n);
   for(int i=0; i<n; ++i) res[i] = a[i]*b[i];
   fwht(res, true);
}
```

6.32 Convolução com CRT

Utiliza o teorema chinês dos restos e duas NTT's para calcular a resposta módulo $mod[0]*mod[1] = 1,097,572,091,361,755,137$. Este número é normalmente grande o suficiente para calcular os valores exatos se as arrays originais tiverem cada elemento menor que aproximadamente 10^6 e $n \leq 2^{20}$. Implementação do teorema chinês dos restos por cortesia do IME.

```
template<typename T>
T modMul(T a, T b, T m) { ... }

//convolution mod 1,097,572,091,361,755,137
void modConv(vector<ll> a, vector<ll> b, vector<ll>
    & res) {
   vector<ll> r0, r1;
   convolution(a, b, r0, 0);
   convolution(a, b, r1, 1);
   ll x, y, s, r, p = mod[0]*mod[1];
   extGcd(mod[0], mod[1], r, s);
   res.resize(r0.size());
   for(int i=0; i<(int)res.size(); i++) {
      res[i] = (modMul((s*mod[1]+p)%p, r0[i], p)
      + modMul((r*mod[0]+p)%p, r1[i], p) + p) % p;
   }
}
```

6.33 Convolução com Decomposição SQRT

Se os números forem menores que aproximadamente 10^6, separa a primeira metade de bits da segunda em cada array e executa 4 FFT's com números menores que aproximadamente 10^3. Isso permite a FFT complexa com double ter precisão suficiente pra calcular de forma exata. Depois basta juntar.

```
#include <cmath>
#define MOD 1000003LL
#define SMOD 1024LL // ~ sqrt(MOD)
typedef long long ll;

void sqrtConv(vector<ll> a, vector<ll> b, vector<ll>
     & c) {
    vector<base> ca[2], cb[2], cc[2][2];
    ca[0].resize(a.size());
    ca[1].resize(a.size());
    for(int i=0; i<(int)a.size(); i++) {
       ca[0][i] = base(a[i] % SMOD, 0);
       ca[1][i] = base(a[i] / SMOD, 0);
    }
    cb[0].resize(b.size());
    cb[1].resize(b.size());
    for(int i=0; i<(int)b.size(); i++) {
       cb[0][i] = base(b[i] % SMOD, 0);
       cb[1][i] = base(b[i] / SMOD, 0);
    }
    for(int l=0; l<2; l++) for(int r=0; r<2; r++)
       convolution(ca[l], cb[r], cc[l][r]);
    c.resize(cc[0][0].size());
    for(int i=0; i<(int)c.size(); i++) {
       c[i] =
       (((ll)round(cc[1][1][i].real()))%MOD*(SMOD*
           SMOD)%MOD)%MOD +
       ((ll)round(cc[0][1][i].real()))%MOD*SMOD%MOD +
       ((ll)round(cc[1][0][i].real()))%MOD*SMOD%MOD +
       ((ll)round(cc[0][0][i].real()))%MOD;
       c[i] %= MOD;
    }
}
```

6.34 Integração pela regra de Simpson

Integração por interporlação quadrática. Erro: $\frac{h^4}{180}(b-a)max_{x\in[a,b]}|f^{(4)}(x)|$, $h=(b-a)/n$.

```
double f(double x) { ... }

double simpson(double a, double b, int n = 1e6) {
   double h = (b - a) / n, s = f(a) + f(b);
   for (int i = 1; i < n; i += 2) s += 4*f(a+h*i);
   for (int i = 2; i < n; i += 2) s += 2*f(a+h*i);
   return s*h/3;
}
```

6.35 Código de Gray

Converte para o cógido de gray, ida $O(1)$ e volta $O(\log n)$. Útil para gerar números consecutivos que diferem por 1 bit. Caso se fixe o número de bits 1 do código, eles saem em ordem a diferirem por uma posição.

```
int gray(int n) { return n ^ (n >> 1); }

int rev_gray(int g) {
   int n = 0;
   for (; g; g>>=1) n ^= g;
   return n;
}
```

6.36 BigInteger em Java

```
import java.util.Scanner;
import java.math.BigInteger;

public final class Main { /* UVa 10925 - Krakovia */
   public static void main(String[] args) {
      Scanner sc = new Scanner(System.in);
      int caseNo = 1;
      while (true) {
         int N = sc.nextInt(), F = sc.nextInt();
         if (N == 0 && F == 0) break;
         BigInteger sum = BigInteger.ZERO;
         for (int i = 0; i < N; i++) {
            BigInteger V = sc.nextBigInteger();
            sum = sum.add(V);
         }
         System.out.println("Bill␣#" +
            (caseNo++) + "␣costs␣" + sum +
         ":␣each␣friend␣should␣pay␣" +
         sum.divide(BigInteger.valueOf(F)));
         System.out.println();
      }
   }
}
```

6.37 Bignum em C++

print imprime o número. *fix* remove os zeros à frente. *str2bignum* converte de string para para bignum. *int2bignum* gera um bignum a partir de um inteiro menor que a base. *bignum2int* só funciona se não der overflow. A divisão por inteiros só funciona para inteiros menores que a base. Soma, subtração, shift left e shift right em $O(n)$, multiplicação em $O(n^2)$. Divisão e resto em uma única operação, é lenta para bases muito grandes. A subtração só funciona para $a \geq b$.

```
#include <vector>
#include <algorithm>
#include <cstring>
using namespace std;

typedef vector<int> bignum;
const int base = 1000*1000*1000;

void print(bignum & a) {
    printf("%d", a.empty() ? 0 : a.back());
    for (int i=(int)a.size()-2; i>=0; --i) {
        printf("%09d", a[i]);
    }
}
void fix(bignum & a) {
    while (a.size() > 1u && a.back() == 0)
        a.pop_back();
}
bool comp(bignum a, bignum b) {
    fix(a); fix(b);
    if (a.size() != b.size()) return a.size() < b.
        size();
    for(int i=(int)a.size()-1; i>=0; i--) {
        if (a[i] != b[i]) return a[i] < b[i];
    }
    return false;
}
void str2bignum(char* s, bignum & a) {
    a.clear();
    for (int i=(int)strlen(s); i>0; i-=9) {
        s[i] = 0;
        a.push_back(atoi(i>=9 ? s+i-9 : s));
    }
    fix(a);
}
void int2bignum(int n, bignum & a) {
    a.clear();
    if (n == 0) a.push_back(0);
    for(; n > 0; n /= base)
        a.push_back(n%base);
}
int bignum2int(bignum & a) {
    int ans = 0, p=1;
    for(int i=0; i<(int)a.size(); i++) {
        ans += a[i]*p; p *= base;
    }
    return ans;
}
void sum(bignum & a, bignum & b, bignum & c) {
    int carry = 0, n = max(a.size(), b.size());
    c.resize(n);
    for (int i=0, ai, bi; i<n; i++) {
        ai = i < (int)a.size() ? a[i] : 0;
        bi = i < (int)b.size() ? b[i] : 0;
        c[i] = carry + ai + bi;
        carry = c[i] / base;
        c[i] %= base;
    }
    if (carry > 0) c.push_back(carry);
    fix(c);
}
void subtract(bignum & a, bignum & b, bignum & c) {
    int carry = 0, n = max(a.size(), b.size());
    c.resize(n);
    for (int i=0, ai, bi; i<n; i++) {
        ai = i < (int)a.size() ? a[i] : 0;
        bi = i < (int)b.size() ? b[i] : 0;
        c[i] = carry + ai - bi;
        carry = c[i] < 0 ? 1 : 0;
        if (c[i] < 0) c[i] += base;
    }
    fix(c);
}
void shiftL(bignum & a, int b, bignum & c) {
    c.resize((int)a.size() + b);
    for(int i=(int)c.size()-1; i>=0; i--) {
        if(i>=b) c[i] = a[i-b];
        else c[i] = 0;
    }
    fix(c);
}
void shiftR(bignum & a, int b, bignum & c) {
    if (((int)a.size()) <= b) {
        c.clear(); c.push_back(0);
        return;
    }
    c.resize((int)a.size() - b);
    for(int i=0; i<(int)c.size(); i++)
        c[i] = a[i+b];
    fix(c);
}
void multiply(int a, bignum & b, bignum & c) {
    int carry = 0, bi;
    c.resize(b.size());
    for (int i=0; i<(int)b.size() || carry; i++) {
        if (i == (int)b.size()) c.push_back(0);
        bi = i < (int)b.size() ? b[i] : 0;
        long long cur = carry + a * 1ll * bi;
        c[i] = int(cur % base);
        carry = int(cur / base);
    }
    fix(c);
}
void multiply(bignum a, bignum b, bignum & c) {
    int n = a.size()+b.size();
    long long carry = 0, acum;
    c.resize(n);
    for (int k=0; k<n || carry; k++) {
        if (k == n) c.push_back(0);
        acum = carry; carry = 0;
        for (int i=0, j=k; i <= k && i<(int)b.size();
             i++, j--) {
            if (j >= (int)b.size()) continue;
            acum += a[i] * 1ll * b[j];
            carry += acum / base;
            acum %= base;
        }
        c[k] = acum;
    }
    fix(c);
}
void divide(bignum & a, int b, bignum & c) {
    int carry = 0;
    c.resize(a.size());
    for (int i=(int)a.size()-1; i>=0; --i) {
        long long cur = a[i] + carry * 1ll * base;
        c[i] = int (cur / b);
        carry = int (cur % b);
    }
    fix(a);
}
```

6.38 A ruína do Apostador

Seja o seguinde jogo: dois jogadores 1 e 2 com n_1 e n_2 moedas, respectivamente jogam um jogo tal que, a cada rodada, existem probabilidade p do jogador 2 transferir uma moeda para 1 (1 ganha a rodada) e $q = 1 - p$ do contrário (2 ganha a rodada). O jogo acaba quando um dos jogadores fica sem moeda (perdedor). A probabilidade de cada um vencer é:

$$P_1 = \frac{n_1}{n_1 + n_2}, P_2 = \frac{n_2}{n_1 + n_2}, p = q \qquad P_1 = \frac{1 - (\frac{q}{p})^{n_1}}{1 - (\frac{q}{p})^{n_1+n_2}}, P_2 = \frac{1 - (\frac{p}{q})^{n_2}}{1 - (\frac{p}{q})^{n_1+n_2}}, p \neq q \tag{6.9}$$

6.39 Teoremas e Fórmulas

- **Desarranjo**: o número $der(n)$ de permutações de n elementos em que nenhum dos elementos fica na posição original é dado por: $der(n) = (n-1)(der(n-1) + der(n-2))$, onde $der(0) = 1$ e $der(1) = 0$.
- **Fórmula de Euler para poliedros convexos**: $V - E + F = 2$, onde F é o número de faces.
- **Círculo de Moser**: o número de peças em que um círculo pode ser divido por cordas ligadas a n pontos tais que não se tem 3 cordas internamente concorrentes é dada por: $g(n) = \binom{n}{4} + \binom{n}{2} + 1 = \frac{1}{24}(n^4 - 6n^3 + 23n^2 - 18n + 24)$.
- **Teorema de Pick**: se I é o número de pontos inteiros dentro de um polígono, A a área do polígono e b o número de pontos inteiros na borda, então $A = i + b/2 - 1$.
- **Teorema de Zeckendorf**: qualquer inteiro positivo pode ser representado pela soma de números de Fibonacci que não inclua dois números consecutivos. Para achar essa soma, usar o algoritmo guloso, sempre procurando o maior número de fibonacci menor que o número.
- **Teorema de Wilson**: um número n é primo se e somente se $(n-1)! \equiv -1 \mod n$
- **Teorema de Euler**: se a e b forem coprimos entre si, então $a^{\phi(b)} \equiv 1 \mod b$ ou $a^n \equiv a^{n\%\phi(b)} \mod b$.
- **Teorema Pequeno de Fermat**: se p é um número primo, então, para qualquer inteiro a, $a^p - a$ é múltiplo de p. Ou seja, $a^p \equiv a \mod p$ ou $a^{p-1} \equiv 1 \mod p$.
- **Último Teorema de Fermat**: para qualquer inteiro $n > 2$, a equação $x^n + y^n = z^n$ não possui soluções inteiras.
- **Teorema de Lagrange**: qualquer inteiro positivo pode ser escrito como a soma de 4 quadrados perfeitos.
- **Conjectura de Goldbach**: qualquer par $n > 2$ pode ser escrito como a soma de dois primos (testada até 4×10^{18}).
- **Conjectura dos primos gêmeos**: existem infinitos primos p tal que $p + 2$ também é primo.
- **Conjectura de Legendre**: para todo n inteiro positivo, existe um primo entre n^2 e $(n+1)^2$.

Capítulo 7

Processamento de Strings

7.1 Função Prefixo: Algoritmo de Knuth-Morris-Pratt (KMP)

String matching em $O(n+m)$. Inicializar a classe com a string a ser procurada e usar match para receber o vector com as posições de matches. Uso alternativo: $b[i]$ é o comprimento do maior prefixo de tamanho menor que i que também é um sufixo dos i primeiros caracteres, $b[0] = -1$.

```
class KMP {
   string P;
   vector<int> b;
   int m;
public:
   KMP(const char* _P) : P(_P) {
      m = P.size(); b.assign(m+1, -1);
      for(int i = 0, j = -1; i < m;) {
         while (j >= 0 && P[i] != P[j]) j = b[j];
         b[++i] = ++j;
      }
   }
   vector<int> match(const char* T) {
      vector<int> ans;
      for (int i=0, j=0, n=strlen(T); i < n;) {
         while (j >= 0 && T[i] != P[j]) j = b[j];
         i++; j++;
         if (j == m) {
            ans.push_back(i - j);
            j = b[j];
         }
      }
      return ans;
   }
};
```

7.2 Rabin-Karp

String matching em $O(m)$, construtor $O(n)$. $hash(i,j) = \sum_{k=i}^{j} s[k]p^{k-i} \mod m$, $O(1)$. Usar hashing para verificar igualdade de strings. Se necessário, fazer com 3 ou 5 pares P e M diferentes. Usar M potência de dois é pedir pra ser hackeado. Abaixo, a probabilidade de colisão por complexidade e M para $n = 10^6$ (tabela por Antti Laaksonen). Observe o paradoxo do aniversário: em uma sala com 23 pessoas, a probabilidade que duas tenham aniversário no mesmo dia é de 50%.

Cenário	Probabilidade	10^3	10^6	10^9	10^{12}	10^{15}	10^{18}
entre duas strings $O(1)$	$1/M$	0.001000	0.000001	0.000000	0.000000	0.000000	0.000000
uma string contra n $O(n)$	$1-(1-1/M)^n$	1.000000	0.632121	0.001000	0.000000	0.000000	0.000000
pares de n strings $O(n^2)$	$1-M!/(M^n(M-n)!)$	1.000000	1.000000	1.000000	0.393469	0.000500	0.000001

```
class RabinKarp {
   ll m;
   vector<int> pw, hsh;
public:
   RabinKarp() {}
   RabinKarp(char str[], ll p, ll _m) : m(_m) {
      int n = strlen(str);
      pw.resize(n); hsh.resize(n);
      hsh[0] = str[0]; pw[0] = 1;
      for(int i = 1; i < n; i++) {
         hsh[i] = (hsh[i-1] * p + str[i]) % m;
         pw[i] = (pw[i-1] * p) % m;
      }
   }
   ll hash(int i, int j) {
      ll ans = hsh[j];
      if (i > 0) ans = (ans - ((hsh[i-1]*1ll*pw[j-i
          +1])%m) + m) % m;
      return ans;
   }
};
```

7.3 Repetend: menor período de uma string

Retorna o menor que k tal que $k|n$ e a string pode ser decomposta em n/k strings iguais.

```
#define MAXN 100009

int repetend(char* s) {
    int n = strlen(s);
    int nxt[n+1];
    nxt[0] = -1;
    for (int i = 1; i <= n; i++) {
        int j = nxt[i - 1];
        while (j >= 0 && s[j] != s[i - 1])
            j = nxt[j];
        nxt[i] = j + 1;
    }
    int a = n - nxt[n];
    if (n % a == 0) return a;
    return n;
}
```

7.4 Suffix Array: Algoritmo de Karp-Miller-Rosenberg (KMR)

suffixArray computa a Suffix Array em $O(n \log n)$. *computeLcp* computa o Longuest Common Prefix em $O(n)$. *lcp*[i] é o tamanho do maior prefixo comum entre *sa*[i] e *sa*[$i-1$]. *lowerBound* computa o menor índice da Suffix Array de um sufixo de s maior que ou igual a t pela ordem lexicográfica em $O(m + \log n)$. *match* computa o número de aparições de t em s em $O(m + \log n)$. CUIDADO: s e t não devem compartilhar o mesmo buffer.

```
int tsa[MAXN], c[MAXN], tc[MAXN], head[MAXN];

void bucket(int sa[], int n, int L) {
   memset(&head, 0, sizeof head);
   for(int i = 0; i < n; i++) head[c[i]]++;
   int maxi = max(300, n), k = 0;
   for(int i = 0; i < maxi; i++)
      swap(k, head[i]), k += head[i];
   for(int i = 0; i < n; i++) {
      int j = (sa[i]-L+n)%n;
      tsa[head[c[j]]++] = j;
   }
   memcpy(sa, tsa, n*sizeof(int));
}

void suffixArray(char str[], int sa[]) {
   int n = strlen(str) + 1;
   for(int i = 0; i < n; i++)
      sa[i] = i, c[i] = str[i];
   bucket(sa, n, 0);
   for(int L = 1; L < n; L <<= 1) {
      bucket(sa, n, L);
      for(int i = 0, cc = -1; i < n; i++)
         tc[sa[i]] = (i == 0 || c[sa[i]] != c[sa[i
             -1]] || c[(sa[i]+L)%n] != c[(sa[i-1]+L)
             %n]) ? ++cc : cc;
      memcpy(c, tc, n*sizeof(int));
   }
   for(int i = 0; i < n-1; i++) sa[i] = sa[i+1];
}

int inv[MAXN];

int computeLcp(char str[], int sa[], int lcp[]) {
   int n = strlen(str), k = 0;
   for(int i = 0; i < n; i++) inv[sa[i]] = i;
   for(int j = 0; j < n; j++) {
      int i = inv[j];
      if (k < 0) k = 0;
      while(i > 0 && str[sa[i]+k] == str[sa[i-1]+k])
          k++;
      lcp[i] = k--;
   }
}

int lowerBound(char s[], char t[], int sa[], int n,
    int m) {
   int hi = n, lo = -1;
   int khi = 0, klo = 0;
   while(hi > lo + 1) {
      int mid = (hi + lo) / 2;
      int i = sa[mid], k = min(klo, khi);
      while(k < m && s[i+k] == t[k]) k++;
      if (k == m || s[i+k] > t[k])
         hi = mid, khi = k;
      else lo = mid, klo = k;
   }
   return hi;
}

int match(char s[], char t[], int sa[], int n, int m
    ) {
   int l = lowerBound(s, t, sa, n, m);
   t[m-1]++;
   int r = lowerBound(s, t, sa, n, m);
   t[m-1]--;
   return r - l;
}
```

7.5 Função Z e Algoritmo Z

Função Z é uma função tal que $z[i]$ é máximo e $str[j] = str[i+j]$, $0 \le j \le z[i]$. O algoritmo Z computa todos os $z[i]$, $0 \le i < n$, em $O(n)$. $z[0] = 0$.

```
void zfunction(char s[], int z[]) {
   int n = strlen(s);
   fill(z, z+n, 0);
   for (int i=1, l=0, r=0; i<n; ++i) {
      if (i <= r) z[i] = min(r-i+1, z[i-l]);
      while (i+z[i] < n && s[z[i]] == s[i+z[i]])
         ++z[i];
      if (i+z[i]-1 > r)
         l = i, r = i+z[i]-1;
   }
}
```

7.6 Algoritmo de Manacher

Usado para achar a maior substring palíndromo. A função *manacher* calcula a array L. $L[i]$ é o máximo valor possível tal que $str[i+j] = str[i-j]$, $0 \leq j \leq L[i]$. Pra calcular os palíndromos pares, basta adicionar '|' entre todos os caracateres e calcular o maior valor de L da string.

```
void manacher(char str[], int L[]) {
   int n = strlen(str), c = 0, r = 0;
   for(int i = 0; i < n; i++) {
      if(i < r && 2*c >= i)
         L[i] = min(L[2*c-i], r-i);
      else L[i] = 0;
      while(i-L[i]-1 >= 0 && i+L[i]+1 < n &&
         str[i-L[i]-1] == str[i+L[i]+1]) L[i]++;
      if(i+L[i]>r) { c=i; r=i+L[i]; }
   }
}
```

7.7 Aho-Corasick

Resolve o problema de achar ocorrências de um dicionário em um texto em $O(n)$, onde n é o comprimento do texto. Pré-processamento: $O(mk)$, onde m é a soma do número de caracteres de todas as palavras do dicionário e k é o tamanho do alfabeto. Cuidado: o número de matches tem pior caso $O(n\sqrt{m})$! Guardar apenas o número de matches, se for o que o problema pedir. Caso o alfabeto seja muito grande, trocar nxt por um *map* ou usar lista de adjacência.

```
#define ALFA 62
#define MAXS 2000009
int nxt[MAXS][ALFA], fail[MAXS], cnt = 0;
vector<ii> pats[MAXS];

class AhoCorasick {
private:
   int root;
   int suffix(int x, int c) {
      while (x != root && nxt[x][c] == 0) x = fail[x
          ];
      return nxt[x][c] ? nxt[x][c] : root;
   }
   int newnode() {
      int x = ++cnt;
      fail[x] = 0; pats[x].clear();
      for(int c = 0; c < ALFA; c++) nxt[x][c] = 0;
      return x;
   }
   inline int reduce(char c) {
      if (c >= 'a' && c <= 'z') return c - 'a';
      if (c >= 'A' && c <= 'Z') return c - 'A' + ('z
          '-'a'+1);
      if (c >= '0' && c <= '9') return c - '0' + 2*(
          'z'-'a'+1);
      return -1;
   }
public:
   AhoCorasick() { root = newnode(); }
   void setfails() {
      queue<int> q;
      int x, y;
      q.push(root);
      while (!q.empty()) {
         x = q.front(); q.pop();
         for (int c = 0; c < ALFA; c++) {
            y = nxt[x][c];
            if (y == 0) continue;
            fail[y] = x == root ? x : suffix(fail[x
                ], c);
            pats[y].insert(pats[y].end(),
               pats[fail[y]].begin(), pats[fail[y]].
                  end());
            q.push(y);
         }
      }
   }
   void insert(const char* s, int id) {
      int len = strlen(s);
      int x = root;
      for (int i = 0; i < len; i++) {
         int & y = nxt[x][reduce(s[i])];
         if (y == 0 || y == root) {
            y = newnode();
         }
         x = y;
      }
      pats[x].push_back(ii(id, len));
   }
   vector<ii> match(const char *s) { //(id, pos)
      int x = root;
      vector<ii> ans;
      for (int i = 0; s[i]; i++) {
         x = suffix(x, reduce(s[i]));
         for(int j = 0; j < (int)pats[x].size(); j
             ++) {
            ii cur = pats[x][j];
            ans.push_back(ii(cur.first, i - cur.
                second + 1));
         }
      }
      return ans;
   }
};
```

7.8 Autômato de Sufixos

Constrói o autômato de sufixos online em $O(n)$ de tempo e $O(nk)$ memória, em que n é a soma dos tamanhos da strings e k é o tamanho do alfabeto. Caso k seja muito grande, trocar nxt por um *map*. Resolve os problemas de string matching com contagem de aparições em $O(m)$, número de substrings diferentes em $O(n)$, maior string repetida em $O(n)$ e maior substring comum em $O(m)$. Os estados terminais são $last, link(last), link(link(last)), \cdots$.

```
#define MAXS 500009 // 2*MAXN
#define ALFA 26

class SuffixAutomaton {
   int len[MAXS], link[MAXS], cnt[MAXS];
   int nxt[MAXS][ALFA], sz, last, root;
   int newnode() {
      int x = ++sz;
      len[x] = 0; link[x] = -1; cnt[x] = 1;
      for(int c = 0; c < ALFA; c++) nxt[x][c] = 0;
      return x;
   }
   inline int reduce(char c) { return c - 'a'; }
public:
   SuffixAutomaton() { clear(); }
   void clear() {
      sz = 0;
      root = last = newnode();
   }
   void insert(const char *s) {
      for(int i = 0; s[i]; i++) extend(reduce(s[i]))
          ;
   }
   void extend(int c) {
      int cur = newnode(), p;
      len[cur] = len[last] + 1;
      for(p = last; p != -1 && !nxt[p][c]; p = link[
          p]) {
         nxt[p][c] = cur;
      }
      if (p == -1) link[cur] = root;
      else {
         int q = nxt[p][c];
         if (len[p] + 1 == len[q]) link[cur] = q;
         else {
            int clone = newnode();
            len[clone] = len[p] + 1;
            for(int i = 0; i < ALFA; i++)
               nxt[clone][i] = nxt[q][i];
            link[clone] = link[q];
            cnt[clone] = 0;
            while(p != -1 && nxt[p][c] == q) {
               nxt[p][c] = clone;
               p = link[p];
            }
            link[q] = link[cur] = clone;
         }
      }
      last = cur;
   }
   bool contains(const char *s) {
      for(int i = 0, p = root; s[i]; i++) {
         int c = reduce(s[i]);
         if (!nxt[p][c]) return false;
         p = nxt[p][c];
      }
      return true;
   }
   long long numDifSubstrings() {
      long long ans = 0;
      for(int i=root+1; i<=sz; i++)
         ans += len[i] - len[link[i]];
      return ans;
   }
   int longestCommonSubstring(const char *s) {
      int cur = root, curlen = 0, ans = 0;
      for(int i = 0; s[i]; i++) {
         int c = reduce(s[i]);
         while(cur != root && !nxt[cur][c]) {
            cur = link[cur];
            curlen = len[cur];
         }
         if (nxt[cur][c]) {
            cur = nxt[cur][c];
            curlen++;
         }
         if (ans < curlen) ans = curlen;
      }
      return ans;
   }
private:
   int deg[MAXS];
public: // chamar computeCnt antes!
   void computeCnt() {
      fill(deg, deg+sz+1, 0);
      for(int i=root+1; i<=sz; i++)
         deg[link[i]]++;
      queue<int> q;
      for(int i=root+1; i<=sz; i++)
         if (deg[i] == 0) q.push(i);
      while(!q.empty()) {
         int i = q.front(); q.pop();
         if (i <= root) continue;
         int j = link[i];
         cnt[j] += cnt[i];
         if ((--deg[j]) == 0) q.push(j);
      }
   }
   int nmatches(const char *s) {
      int p = root;
      for(int i = 0; s[i]; i++) {
         int c = reduce(s[i]);
         if (!nxt[p][c]) return 0;
         p = nxt[p][c];
      }
      return cnt[p];
   }
   int longestRepeatedSubstring(int times) {
      int ans = 0;
      for(int i=root; i<=sz; i++) {
         if (cnt[i] >= times && ans < len[i]) {
            ans = len[i];
         }
      }
      return ans;
   }
};
```

7.9 Inaccurate String Matching com FFT

Seja n o tamanho de T (texto) e m o tamanho de P (padrão). Realiza o a contagem de matches em $O(nk \log n)$, em k é o tamanho do alfabeto. $match[i]$ = o número de caracteres da substring $T[i \dots i+m-1]$ que batem com P. Para $i > n-m$, ele considera que a string P rotaciona sobre T. Retorna 0 se $m > n$.

```
/*codigo de FFT aqui*/
#define ALFA 5

int reduce(char c) {
   if (c >= 'a' && c <= 'z') return c-'a';
   if (c >= 'A' && c <= 'Z') return c-'A'+'z';
   return -1;
}

void matchfft(const char *T, const char *P, int *
    match) {
   int n = strlen(T), m = strlen(P);
   memset(match, 0, n*sizeof(int));
   if (m > n) return;
   vector<base> Sa[ALFA], Sb[ALFA], M[ALFA];
   for(int c = 0; c<ALFA; c++) {
      Sa[c].resize(n); Sa[c].assign(n, 0);
      Sb[c].resize(n); Sb[c].assign(n, 0);
   }
   for(int i=0; i<n; i++) Sa[reduce(T[i])][i] = 1;
   for(int i=0; i<m; i++) Sb[reduce(P[i])][i] = 1;
   for(int c = 0; c<ALFA; c++) {
      reverse(Sb[c].begin(), Sb[c].end());
      circularConv(Sa[c], Sb[c], M[c]);
   }
   for(int i=0; i<n; i++) {
      for(int c=0; c<ALFA; c++) {
         match[(i+1)%n] += int(M[c][i].real() + 0.5)
             ;
      }
   }
}
```

7.10 Autômato de KMP

A *struct trans* é gerada a partir de um padrão $P, m = |P|$, e de uma *string* s. Ela representa as transições ($to[i]$) e número de *matchings* ($match[i]$) a partir do estado i se for adicionada a *string* s ao final da *string*. Ou seja, se ao final da *string* acabamos no estado i e adicionamos a *string* s, iremos para o estado $to[i]$ e adicionaremos $match[i]$ *matchings*. O estado i (0-indexed) é o estado após já se terem os primeiros i caracteres de P pareados. A transformação de um caractere é gerada pelo gerador KMP em $O(m)$. O operador + serve para se obter as transformações de estado ao se adicionarem duas *strings* em sequência. Útil para se usar com *Segment Tree* com queries de número de matchings em uma substring em $O(m \log n)$.

```
#define MAXM 109

struct trans {
   int m, to[MAXM], match[MAXM];
   trans(int _m = -1) : m(_m) {}
};

trans operator +(trans a, trans b) {
   if (b.m < 0) return a;
   if (a.m < 0) return b;
   //assert(a.m == b.m);
   trans c(a.m);
   for(int i = 0; i < c.m; i++) {
      c.to[i] = b.to[a.to[i]];
      c.match[i] = a.match[i] + b.match[a.to[i]];
   }
   return c;
}

struct KMP {
   char P[MAXM];
   int b[MAXM], m;
   void build(const char* _P) {
      m = strlen(_P);
      strcpy(P, _P);
      memset(&b, -1, sizeof b);
      for(int i = 0, j = -1; i < m;) {
         while (j >= 0 && P[i] != P[j]) j = b[j];
         b[++i] = ++j;
      }
   }
   trans get(char c) {
      trans ans(m);
      for(int i = 0, j, cnt; i < m; i++) {
         j = i; cnt = 0;
         while(j >= 0 && c != P[j]) j = b[j];
         j++;
         if (j == m) cnt++, j = b[j];
         ans.to[i] = j;
         ans.match[i] = cnt;
      }
      return ans;
   }
};
```

7.11 Palindromic Tree

Palindromic Tree em $O(n)$. $root1$ é a raíz de tamanho -1 (palíndromos ímpares). $root2$ é a raíz de tamanho 0 (palíndromos pares). $len[u]$ é o tamanho do palíndromo no nó u. $nxt[u][c]$ aponta para um nó que representa um palíndromo de tamanho $len[u] + 2$ e começa e acaba com c. $link[u]$ aponta pro maior sufixo menor que o palíndromo de u que também é palíndromo. $extend(c)$ adiciona c ao final da string em $O(1)$ amortizado e retorna quantos palíndromos novos não distintos foram adicionados. O número de palíndromos distintos é o número de nós não triviais. Ao resetar cnt, é necessário limpar nxt.

```
#define MAXS 1000009
#define ALFA 26

int nxt[MAXS][ALFA], len[MAXS];
int link[MAXS], num[MAXS];
int cnt = 0; //limpar nxt se me resetar

class PalindromicTree {
private:
   string s;
   int suff;
   char reduce(char c) { return c-'a'; }
   int getlink(int u, int pos) {
      for(; true; u = link[u]) {
         int st = pos - 1 - len[u];
         if (st >= 0 && s[st] == s[pos])
            return u;
      }
   }
public:
   int root1, root2;
   PalindromicTree() {
      root1 = ++cnt; root2 = ++cnt;
      suff = root2;
      len[root1] = -1; link[root1] = root1;
      len[root2] = 0; link[root2] = root1;
   }
   int extend(char c) {
      int pos = s.size(); s.push_back(c);
      c = reduce(c);
      int u = getlink(suff, pos);
      if (nxt[u][c]) {
         suff = nxt[u][c];
         return 0;
      }
      int v = suff = ++cnt;
      len[v] = len[u] + 2;
      nxt[u][c] = v;
      if (len[v] == 1) {
         link[v] = root2;
         return num[v] = 1;
      }
      u = getlink(link[u], pos);
      link[v] = nxt[u][c];
      return num[v] = 1 + num[link[v]];
   }
};
```

Capítulo 8

Geometria Computacional

8.1 Ponto 2D e segmentos de reta

Ponto com double em 2D com algumas funcionalidades: distância, produto interno, produto vetorial (componente z), teste counter-clockwise, teste de colinearidade, rotação em relação ao centro do plano, projeção de u sobre v, ponto dentro de segmento de reta, intersecção de retas, teste de paralelidade, teste de intersecção de segmentos de reta, ponto mais próximo ao segmento de reta.

```
#define EPS 1e-9

struct point {
   double x, y;
   point() { x = y = 0.0; }
   point(double _x, double _y) : x(_x), y(_y) {}
   double norm() { return hypot(x, y); }
   point normalized() {
      return point(x,y)*(1.0/norm());
   }
   double angle() { return atan2(y, x); }
   double polarAngle() {
      double a = atan2(y, x);
      return a < 0 ? a + 2*acos(-1.0) : a;
   }
   bool operator < (point other) const {
      if (fabs(x - other.x) > EPS) return x < other.
          x;
      else return y < other.y;
   }
   bool operator == (point other) const {
      return (fabs(x - other.x) < EPS && (fabs(y -
          other.y) < EPS));
   }
   point operator +(point other) const {
      return point(x + other.x, y + other.y);
   }
   point operator -(point other) const {
      return point(x - other.x, y - other.y);
   }
   point operator *(double k) const {
      return point(x*k, y*k);
   }
};
double dist(point p1, point p2) {
   return hypot(p1.x - p2.x, p1.y - p2.y);
}
double inner(point p1, point p2) {
   return p1.x*p2.x + p1.y*p2.y;
}
double cross(point p1, point p2) {
   return p1.x*p2.y - p1.y*p2.x;
}
bool ccw(point p, point q, point r) {
   return cross(q-p, r-p) > 0;
}
bool collinear(point p, point q, point r) {
   return fabs(cross(p-q, r-p)) < EPS;
}
point rotate(point p, double rad) {
   return point(p.x * cos(rad) - p.y * sin(rad),
   p.x * sin(rad) + p.y * cos(rad));
}
double angle(point a, point o, point b) {
   return acos(inner(a-o, b-o) / (dist(o,a)*dist(o,b
       )));
}
point proj(point u, point v) {
   return v*(inner(u,v)/inner(v,v));
}
bool between(point p, point q, point r) {
   return collinear(p, q, r) && inner(p - q, r - q)
       <= 0;
}
point lineIntersectSeg(point p, point q, point A,
    point B) {
   double c = cross(A-B, p-q);
   double a = cross(A, B);
   double b = cross(p, q);
   return ((p-q)*(a/c)) - ((A-B)*(b/c));
}
bool parallel(point a, point b) {
   return fabs(cross(a, b)) < EPS;
}
bool segIntersects(point a, point b, point p, point
    q) {
   if (parallel(a-b, p-q)) {
      return between(a, p, b) || between(a, q, b)
          || between(p, a, q) || between(p, b, q);
   }
   point i = lineIntersectSeg(a, b, p, q);
   return between(a, i, b) && between(p, i, q);
}
point closestToLineSegment(point p, point a, point b
    ) {
   double u = inner(p-a, b-a) / inner(b-a, b-a);
   if (u < 0.0) return a;
   if (u > 1.0) return b;
   return a + ((b-a)*u);
}
```

8.2 Círculo 2D

Círculo no plano 2D com algumas funcionalidades: área, comprimento de corda, área do setor, teste de intersecção com outro círculo, teste de ponto, pontos de retas tangentes (se o ponto estiver dentro *asin* retorna *nan*), circumcírculo e incírculo (divisão por zero se os pontos forem colineares).

```
struct circle{
   point c;
   double r;
   circle() { c = point(); r = 0; }
   circle(point _c, double _r) : c(_c), r(_r) {}
   double area() { return acos(-1.0)*r*r; }
   double chord(double rad) { return  2*r*sin(rad
       /2.0); }
   double sector(double rad) { return 0.5*rad*area()
       /acos(-1.0); }
   bool intersects(circle other) {
      return dist(c, other.c) < r + other.r;
   }
   bool contains(point p) { return dist(c, p) <= r +
        EPS; }
   pair<point, point> getTangentPoint(point p) {
      double d1 = dist(p, c), theta = asin(r/d1);
      point p1 = rotate(c-p,-theta);
      point p2 = rotate(c-p,theta);
      p1 = p1*(sqrt(d1*d1-r*r)/d1)+p;
      p2 = p2*(sqrt(d1*d1-r*r)/d1)+p;
      return make_pair(p1,p2);
   }
   vector< pair<point,point> > getTangentSegs(circle
       other) {
      vector<pair<point, point> > ans;
      double d = dist(other.c, c);
      double dr = abs(r - other.r), sr = r + other.r
          ;
      if (dr >= d) return ans;
      double u = acos(dr / d);
      point dc1 = ((other.c - c).normalized())*r;
      point dc2 = ((other.c - c).normalized())*other
          .r;
      ans.push_back(make_pair(c + rotate(dc1, u),
          other.c + rotate(dc2, u)));
      ans.push_back(make_pair(c + rotate(dc1, -u),
          other.c + rotate(dc2, -u)));
      if (sr >= d) return ans;
      double v = acos(sr / d);
      dc2 = ((c - other.c).normalized()) * other.r;
      ans.push_back({c + rotate(dc1, v), other.c +
          rotate(dc2, v)});
      ans.push_back({c + rotate(dc1, -v), other.c +
          rotate(dc2, -v)});
      return ans;
   }
   pair<point, point> getIntersectionPoints(circle
       other){
      assert(intersects(other));
      double d = dist(c, other.c);
      double u = acos((other.r*other.r + d*d - r*r)
          / (2*other.r*d));
      point dc = ((other.c - c).normalized()) * r;
      return make_pair(c + rotate(dc, u), c + rotate
          (dc, -u));
   }
};

circle circumcircle(point a, point b, point c) {
   circle ans;
   point u = point((b-a).y, -(b-a).x);
   point v = point((c-a).y, -(c-a).x);
   point n = (c-b)*0.5;
   double t = cross(u,n)/cross(v,u);
   ans.c = ((a+c)*0.5) + (v*t);
   ans.r = dist(ans.c, a);
   return ans;
}

int insideCircle(point p, circle c) {
   if (fabs(dist(p , c.c) - c.r)<EPS) return 1;
   else if (dist(p , c.c) < c.r) return 0;
   else return 2;
} //0 = inside/1 = border/2 = outside

circle incircle( point p1, point p2, point p3 ) {
    double m1=dist(p2, p3);
    double m2=dist(p1, p3);
    double m3=dist(p1, p2);
    point c = (p1*m1+p2*m2+p3*m3)*(1/(m1+m2+m3));
    double s = 0.5*(m1+m2+m3);
    double r = sqrt(s*(s-m1)*(s-m2)*(s-m3))/s;
    return circle(c, r);
}
```

8.3 Grande Círculo

Dado o raio da Terra e as coordenadas em latitude e longitude de dois pontos p e q, retorna o ângulo pOq. Retorna a distância mínima de viagem pela superfície.

```
double gcTheta(double pLat, double pLong, double
    qLat, double qLong) {
   pLat *= PI / 180.0; pLong *= PI / 180.0; //
       convert degree to radian
   qLat *= PI / 180.0; qLong *= PI / 180.0;
   return acos(cos(pLat)*cos(pLong)*cos(qLat)*cos(
       qLong) +
      cos(pLat)*sin(pLong)*cos(qLat)*sin(qLong) +
      sin(pLat)*sin(qLat));
}

double gcDistance(double pLat, double pLong, double
    qLat, double qLong, double radius) {
   return radius*gcTheta(pLat, pLong, qLat, qLong);
}
```

8.4 Triângulo 2D

```
struct triangle{
   point a, b, c;
   triangle() { a = b = c = point(); }
   triangle(point _a, point _b, point _c) : a(_a), b
       (_b), c(_c) {}
   double perimeter() { return dist(a,b) + dist(b,c)
       + dist(c,a); }
   double semiPerimeter() { return perimeter()/2.0;
       }
   double area() {
      double s = semiPerimeter(), ab = dist(a,b),
         bc = dist(b,c), ca = dist(c,a);
      return sqrt(s*(s-ab)*(s-bc)*(s-ca));
   }
   double rInCircle() {
      return area()/semiPerimeter();
   }
   circle inCircle() {
      return incircle(a,b,c);
   }
   double rCircumCircle() {
      return dist(a,b)*dist(b,c)*dist(c,a)/(4.0*area
          ());
   }
   circle circumCircle() {
      return circumcircle(a,b,c);
   }
    int isInside(point p) {
      double u = cross(b-a,p-a)*cross(b-a,c-a);
      double v = cross(c-b,p-b)*cross(c-b,a-b);
      double w = cross(a-c,p-c)*cross(a-c,b-c);
      if (u > 0.0 && v > 0.0 && w > 0.0) return 0;
      if (u < 0.0 || v < 0.0 || w < 0.0) return 2;
      else return 1;
   } //0 = inside/ 1 = border/ 2 = outside
};

double rInCircle(point a, point b, point c) {
   return triangle(a,b,c).rInCircle();
}

double rCircumCircle(point a, point b, point c) {
   return triangle(a,b,c).rCircumCircle();
}

int isInsideTriangle(point a, point b, point c,
    point p) {
   return triangle(a,b,c).isInside(p);
} //0 = inside/ 1 = border/ 2 = outside
```

8.5 Polígono 2D

```
typedef vector<point> polygon;

double signedArea(polygon & P) {
   double result = 0.0;
   int n = P.size();
   for (int i = 0; i < n; i++) {
      result += cross(P[i], P[(i+1)%n]);
   }
   return result / 2.0;
}

int leftmostIndex(vector<point> & P) {
   int ans = 0;
   for(int i=1; i<(int)P.size(); i++) {
      if (P[i] < P[ans]) ans = i;
   }
   return ans;
}

polygon make_polygon(vector<point> P) {
   if (signedArea(P) < 0.0) reverse(P.begin(), P.end
       ());
   int li = leftmostIndex(P);
   rotate(P.begin(), P.begin()+li, P.end());
   return P;
}

double perimeter(polygon & P) {
   double result = 0.0;
   int n = P.size();
   for (int i = 0; i < n; i++) result += dist(P[i],
       P[(i+1)%n]);
   return result;
}

double area(polygon & P) {
   return fabs(signedArea(P));
}

bool isConvex(polygon & P) {
   int n = (int)P.size();
   if (n < 3) return false;
   bool left = ccw(P[0], P[1], P[2]);
   for (int i = 1; i < n; i++) {
      if (ccw(P[i], P[(i+1)%n], P[(i+2)%n]) != left)
         return false;
   }
   return true;
}

bool inPolygon(polygon & P, point p) {
   if (P.size() == 0u) return false;
   double sum = 0.0;
   int n = P.size();
   for (int i = 0; i < n; i++) {
      if (P[i] == p || between(P[i], p, P[(i+1)%n]))
          return true;
      if (ccw(p, P[i], P[(i+1)%n])) sum += angle(P[i
          ], p, P[(i+1)%n]);
      else sum -= angle(P[i], p, P[(i+1)%n]);
   }
   return fabs(fabs(sum) - 2*acos(-1.0)) < EPS;
}

polygon cutPolygon(polygon & P, point a, point b) {
   vector<point> R;
   double left1, left2;
   int n = P.size();
   for (int i = 0; i < n; i++) {
      left1 = cross(b-a, P[i]-a);
      left2 = cross(b-a, P[(i+1)%n]-a);
      if (left1 > -EPS) R.push_back(P[i]);
      if (left1 * left2 < -EPS)
         R.push_back(lineIntersectSeg(P[i], P[(i+1)%
             n], a, b));
   }
   return make_polygon(R);
}
```

8.6 Convex Hull

Dado um conjunto de pontos, retorna o menor polígono que contém todos os pontos em $O(n \log n)$. Caso precise considerar os pontos no meio de uma aresta, trocar o teste *ccw* para ≥ 0. CUIDADO: Se todos os pontos forem colineares, vai dar RTE.

```
point pivot(0, 0);

bool angleCmp(point a, point b) {
   if (collinear(pivot, a, b))
      return inner(pivot-a, pivot-a) < inner(pivot-b
          , pivot-b);
   return cross(a-pivot, b-pivot) >= 0;
}

polygon convexHull(vector<point> P) {
   int i, j, n = (int)P.size();
   if (n <= 2) return P;
   int P0 = leftmostIndex(P);
   swap(P[0], P[P0]);
   pivot = P[0];
   sort(++P.begin(), P.end(), angleCmp);
   vector<point> S;
   S.push_back(P[n-1]);
   S.push_back(P[0]);
   S.push_back(P[1]);
   for(i = 2; i < n;) {
      j = (int)S.size()-1;
      if (ccw(S[j-1], S[j], P[i])) S.push_back(P[i
          ++]);
      else S.pop_back();
   }
   reverse(S.begin(), S.end());
   S.pop_back();
   reverse(S.begin(), S.end());
   return S;
}
```

8.7 Ponto dentro de polígono convexo

Dado um polígono convexo no sentido horário, verifica se o ponto está dentro (inclui borda) em $O(\log n)$.

```
bool insideConvex(polygon &P, point q) {
   int i = 1, j = P.size()-1, m;
   if (cross(P[i]-P[0], P[j]-P[0]) < -EPS)
      swap(i, j);
   while(abs(j-i) > 1) {
      int m = (i+j)/2;
      if (cross(P[m]-P[0], q-P[0]) < 0) j = m;
      else i = m;
   }
   return isInsideTriangle(P[0], P[i], P[j], q) !=
       2;
}
```

8.8 Soma de Minkowski

Determina o polígono que contorna a soma de Minkowski de duas regiões delimitadas por polígonos regulares. A soma de Minkowski de dois conjuntos de pontos A e B é o conjunto $C = \{c \in R^2 | c = a + b, a \in A, b \in B\}$. Algumas aplicações interessantes:

- Para verificar se A e B possuem intersecção, basta verificar se $(0,0) \in minkowski(A, -B)$.
- $(1/n) * minkowski(A_1, A_2, ..., A_n)$ representa todos os baricentros possíveis de pontos em $A_1, A_2, ..., A_n$.

```
polygon minkowski(polygon & A, polygon & B) {
   polygon P; point v1, v2;
   int n1 = A.size(), n2 = B.size();
   P.push_back(A[0]+B[0]);
   for(int i = 0, j = 0; i < n1 || j < n2;) {
      v1 = A[(i+1)%n1]-A[i%n1];
      v2 = B[(j+1)%n2]-B[j%n2];
      if (j == n2 || cross(v1, v2) > EPS) {
         P.push_back(P.back() + v1); i++;
      }
      else if (i == n1 || cross(v1, v2) < -EPS) {
         P.push_back(P.back() + v2); j++;
      }
      else {
         P.push_back(P.back() + (v1+v2));
         i++; j++;
      }
   }
   P.pop_back();
   return P;
}
```

8.9 Comparador polar

Função para inteiros: *polarCmp* para $x, y \approx 10^9$.

```
bool polarCmp(point a, point b) {
   if (b.y*a.y > 0) return cross(a, b) > 0;
   else if (b.y == 0 && b.x > 0) return false;
   else if (a.y == 0 && a.x > 0) return true;
   else return b.y < a.y;
}
```

8.10 Triangulação de Delaunay

Execução $O(n \log^2 n)$ com alto overhead. $\approx 2.5s$ para $n = 10^5$. *adj* é a lista de adjacência dos nós, *tri* são os triângulos.

```
#include <set>
#include <algorithm>
#define MAXN 200309

struct truple {
   int a, b, c;
   truple(int _a, int _b, int _c) :
      a(_a), b(_b), c(_c) {
      if (a > b) swap(a, b);
      if (a > c) swap(a, c);
      if (b > c) swap(b, c);
   }
};
bool operator < (truple x, truple y) {
   if (x.a != y.a) return x.a < y.a;
   if (x.b != y.b) return x.b < y.b;
   if (x.c != y.c) return x.c < y.c;
   return false;
}

namespace Delaunay {
vector< set<int> > adj;
vector<point> P;
set<truple> tri;
void add_edge(int u, int v, int w = -1) {
   adj[u].insert(v); adj[v].insert(u);
   if (w >= 0) tri.insert(truple(u, v, w));
}
void del_edge(int u, int v) {
   adj[u].erase(v); adj[v].erase(u);
}
void brute(int l, int r) {
   if (r-l == 2 && collinear(P[l],P[l+1],P[r])) {
      add_edge(l, l+1); add_edge(l+1, r);
      return;
   }
   for(int u = l; u <= r; u++)
      for(int v = u+1; v <= r; v++)
         add_edge(u, v);
   if (r-l == 2) tri.insert(truple(l,l+1,r));
}
double theta[MAXN];
bool comp(int u, int v) {
   return theta[u] < theta[v];
}
vector< vector<int> > g;
bool right; point base;
void computeG(int u, int r = -1) {
   if (!g[u].empty()) return;
   if (r >= 0) adj[u].erase(r);
   g[u] = vector<int>(adj[u].begin(), adj[u].end());
   if (r >= 0) adj[u].insert(r);
   for(int i = 0; i < int(g[u].size()); i++) {
      double co = inner(base, P[g[u][i]]-P[u]);
      double si = cross(base, P[g[u][i]]-P[u]);
      theta[g[u][i]] = atan2(si, co);
   }
   sort(g[u].begin(), g[u].end(), comp);
   if (right) reverse(g[u].begin(), g[u].end());
}
int getNext(int u, int a, int b) {
   right = (u == b);
   base = (right ? P[b]-P[a] : P[a]-P[b]);
   computeG(u, a + b - u);
   int ans = -1, w;
   while(!g[u].empty()) {
      int j = g[u].size() - 1;
      ans = g[u][j];
      if (right && cross(base, P[ans]-P[b]) > -EPS)
         return -1;
      if (!right && cross(base, P[ans]-P[a]) < EPS)
         return -1;
      circle c = circumcircle(P[a],P[b],P[ans]);
      if (g[u].size() > 1u && dist(c.c, P[w = g[u][j
          -1]]) < c.r - EPS) {
         del_edge(u, ans); g[u].pop_back();
         tri.erase(truple(u, w, ans));
      }
      else break;
   }
   return ans;
}
int moveEdge(int & u, int a, int b) {
   right = (b == u); int v = a+b-u;
   for(set<int>::iterator it = adj[u].begin(); it !=
       adj[u].end(); it++) {
      int nu = *it;
      double cr = cross(P[u]-P[v], P[nu]-P[v]);
      if (right && cr > EPS) return u = nu;
      if (!right && cr < -EPS) return u = nu;
      if (fabs(cr) < EPS && dist(P[nu], P[v]) < dist
          (P[u], P[v])) return u = nu;
   }
   return -1;
}
void delaunay(int l, int r) {
   if (r - l < 3) { brute(l, r); return; }
   int mid = (r + l) / 2;
   delaunay(l, mid); delaunay(mid + 1, r);
   int u = l, v = r, nu, nv;
   double du, dv;
   do {
      nu = moveEdge(u, u, v);
      nv = moveEdge(v, u, v);
   } while (nu != -1 || nv != -1);
   g[u].clear(); g[v].clear();
   add_edge(u, v);
   while(true) {
      nu = getNext(u, u, v); nv = getNext(v, u, v);
      if (nu == -1 && nv == -1) break;
      if (nu != -1 && nv != -1) {
         point nor = point(P[u].y-P[v].y, P[v].x-P[u
             ].x);
         circle cu = circumcircle(P[u],P[v],P[nu]);
         circle cv = circumcircle(P[u],P[v],P[nv]);
         du = inner(cu.c-P[u], nor);
         dv = inner(cv.c-P[v], nor);
      }
      else du = 1, dv = 0;
      if (nu == -1 || du < dv) {
         add_edge(u, nv, v); g[v = nv].clear();
      }
      else if (nv == -1 || du >= dv) {
         add_edge(nu, v, u); g[u = nu].clear();
      }
   }
}
vector< set<int> > compute(vector<point> & Q) {
   sort(Q.begin(), Q.end());
   int n = (P = Q).size();
   adj.clear(); adj.assign(n, set<int>());
   g.clear(); g.assign(n, vector<int>());
   tri.clear();
   delaunay(0, n-1);
   return adj;
}
set<truple> getTriangles() { return tri; }
};
```

8.11 Intersecção de polígonos

Intersecção de dois polígons convexos em $O(nm \log(n+m))$.

```
polygon intersect(polygon & A, polygon & B) {
   polygon P;
   int n = A.size(), m = B.size();
   for (int i = 0; i < n; i++) {
      if (inPolygon(B, A[i])) P.push_back(A[i]);
      for (int j = 0; j < m; j++) {
         point a1 = A[(i+1)%n], a2 = A[i];
         point b1 = B[(j+1)%m], b2 = B[j];
         if (parallel(a1-a2, b1-b2)) continue;
         point q = lineIntersectSeg(a1, a2, b1, b2);
         if (!between(a1, q, a2)) continue;
         if (!between(b1, q, b2)) continue;
         P.push_back(q);
      }
   }
   for (int i = 0; i < m; i++){
      if (inPolygon(A, B[i])) P.push_back(B[i]);
   }
   set<point> inuse; //Remove duplicates
   int sz = 0;
   for (int i = 0; i < (int)P.size(); ++i) {
      if (inuse.count(P[i])) continue;
      inuse.insert(P[i]);
      P[sz++] = P[i];
   }
   P.resize(sz);
   if (!P.empty()) {
      pivot = P[0];
      sort(P.begin(), P.end(), angleCmp);
   }
   return P;
}
```

8.12 Minimum Enclosing Circle

Computa o círculo de raio mínimo que contém um conjunto de pontos. Complexidade: *expected* $O(n)$.

```
circle minimumCircle(vector<point> p) {
   random_shuffle(p.begin(), p.end());
   circle C = circle(p[0], 0.0);
   for(int i = 0; i < (int)p.size(); i++) {
      if (C.contains(p[i])) continue;
      C = circle(p[i], 0.0);
      for(int j = 0; j < i; j++) {
         if (C.contains(p[j])) continue;
         C = circle((p[j] + p[i])*0.5, 0.5*dist(p[j
             ], p[i]));
         for(int k = 0; k < j; k++) {
            if (C.contains(p[k])) continue;
            C = circumcircle(p[j], p[i], p[k]);
         }
      }
   }
   return C;
}
```

8.13 Intersecção de semi-planos

Computa a intersecção de semi-planos em $O(n \log n)$.

```
typedef pair<point, point> halfplane;

point dir(halfplane h) {
   return h.second - h.first;
}
bool belongs(halfplane h, point a) {
   return cross(dir(h), a - h.first) > EPS;
}
bool hpcomp(halfplane ha, halfplane hb) {
   point a = dir(ha), b = dir(hb);
   if (parallel(a, b) && inner(a, b) > EPS)
      return cross(b, ha.first - hb.first) < -EPS;
   if (b.y*a.y > EPS) return cross(a, b) > EPS;
   else if (fabs(b.y) < EPS && b.x > EPS) return
       false;
   else if (fabs(a.y) < EPS && a.x > EPS) return
       true;
   else return b.y < a.y;
}
polygon intersect(vector<halfplane> H, double W =
    INF) {
   H.push_back(halfplane(point(-W,-W),point(W,-W)));
   H.push_back(halfplane(point(W,-W),point(W,W)));
   H.push_back(halfplane(point(W,W),point(-W,W)));
   H.push_back(halfplane(point(-W,W),point(-W,-W)));
   sort(H.begin(), H.end(), hpcomp);
   int i = 0;
   while(parallel(dir(H[0]), dir(H[i]))) i++;
   deque<point> P;
   deque<halfplane> S;
   S.push_back(H[i-1]);
   for(; i < (int)H.size(); i++) {
      while(!P.empty() && !belongs(H[i], P.back()))
         P.pop_back(), S.pop_back();
      point df = dir(S.front()), di = dir(H[i]);
      if (P.empty() && cross(df, di) < EPS)
         return polygon();
      P.push_back(lineIntersectSeg(H[i].first, H[i].
          second,
          S.back().first, S.back().second));
      S.push_back(H[i]);
   }
   while(!belongs(S.back(), P.front()) ||
      !belongs(S.front(), P.back())) {
      while(!belongs(S.back(), P.front()))
         P.pop_front(), S.pop_front();
      while(!belongs(S.front(), P.back()))
         P.pop_back(), S.pop_back();
   }
   P.push_back(lineIntersectSeg(S.front().first,
      S.front().second, S.back().first, S.back().
          second));
   return polygon(P.begin(), P.end());
}
```

8.14 Ponto 3D

```
#include <cstdio>
#include <cmath>
#define EPS 1e-9

struct point {
   double x, y, z;
   point() { x = y = z = 0.0; }
   point(double _x, double _y, double _z) : x(_x), y
       (_y), z(_z) {}
   double norm() { return sqrt(x*x + y*y + z*z); }
   point normalized() {
      return point(x,y,z)*(1.0/norm());
   }
   bool operator < (point other) const {
      if (fabs(x - other.x) > EPS)
         return x < other.x;
      else if (fabs(y - other.y) > EPS)
         return y < other.y;
      else return z < other.z;
   }
   bool operator == (point other) const {
      return (fabs(x - other.x) < EPS && fabs(y -
          other.y) < EPS && fabs(z - other.z) < EPS)
          ;
   }
   point operator +(point other) const{
      return point(x + other.x, y + other.y, z +
          other.z);
   }
   point operator -(point other) const{
      return point(x - other.x, y - other.y, z -
          other.z);
   }
   point operator *(double k) const{
      return point(x*k, y*k, z*k);
   }
};
double dist(point p1, point p2) {
   return (p1-p2).norm();
}
double inner(point p1, point p2) {
   return p1.x*p2.x + p1.y*p2.y + p1.z*p2.z;
}
point cross(point p1, point p2) {
   point ans;
   ans.x = p1.y*p2.z - p1.z*p2.y;
   ans.y = p1.z*p2.x - p1.x*p2.z;
   ans.z = p1.x*p2.y - p1.y*p2.x;
   return ans;
}
bool collinear(point p, point q, point r) {
   return cross(p-q, r-p).norm() < EPS;
}
double angle(point a, point o, point b) {
   return acos(inner(a-o, b-o) / (dist(o,a)*dist(o,b
       )));
}
point proj(point u, point v) {
   return v*(inner(u,v)/inner(v,v));
}
```

8.15 Triângulo 3D

```
struct triangle{
   point a, b, c;
   triangle() { a = b = c = point(); }
   triangle(point _a, point _b, point _c) : a(_a), b
       (_b), c(_c) {}
   double perimeter() { return dist(a,b) + dist(b,c)
        + dist(c,a); }
   double semiPerimeter() { return perimeter()/2.0;
       }
   double area() {
      double s = semiPerimeter(), ab = dist(a,b),
         bc = dist(b,c), ca = dist(c,a);
      return sqrt(s*(s-ab)*(s-bc)*(s-ca));
   }
   double rInCircle() {
      return area()/semiPerimeter();
   }
   double rCircumCircle() {
      return dist(a,b)*dist(b,c)*dist(c,a)/(4.0*area
          ());
   }
   point normalVector() {
      return cross(y-x, z-x).normalized();
   }
   int isInside(point p) {
      point n = normalVector();
      double u = proj(cross(b-a,p-a), n).normalized
          ()*proj(cross(b-a,c-a), n).normalized();
      double v = proj(cross(c-b,p-b), n).normalized
          ()*proj(cross(c-b,a-b), n).normalized();
      double w = proj(cross(a-c,p-c), n).normalized
          ()*proj(cross(a-c,b-c), n).normalized();
      if (u > 0.0 && v > 0.0 && w > 0.0) return 0;
      else if (u < 0.0 || v < 0.0 || w < 0.0) return
           2;
      else return 1;
   } //0 = inside/ 1 = border/ 2 = outside
   int isProjInside(point p) {
      return isInside(p + proj(a-p, normalVector()))
          ;
   } //0 = inside/ 1 = border/ 2 = outside
};

double rInCircle(point a, point b, point c) {
   return triangle(a,b,c).rInCircle();
}

double rCircumCircle(point a, point b, point c) {
   return triangle(a,b,c).rCircumCircle();
}

int isProjInsideTriangle(point a, point b, point c,
    point p) {
   return triangle(a,b,c).isProjInside(p);
} //0 = inside/ 1 = border/ 2 = outside
```

8.16 Linha 3D

Desta vez a linha é implementada com um ponto de referência e um vetor base. *distVector* é um vetor que é perpendicular a ambas as linhas e tem como comprimento a distância entre elas. *distVector* é a "ponte"de a a b de menor caminho entre as duas linhas. *distVectorBasePoint* é o ponto da linha a de onde sai o *distVector*. *distVectorEndPoint* é o ponto na linha b onde chega a ponte.

```
struct line{
    point r;
    point v;
    line(point _r, point _v) {
        v = _v; r = _r;
    }
    bool operator == (line other) const{
        return fabs(cross(r-other.r, v).norm()) <
            EPS && fabs(cross(r-other.r, other.v).
            norm()) < EPS;
    }
};

point distVector(line l, point p) {
    point dr = p - l.r;
    return dr - proj(dr, l.v);
}

point distVectorBasePoint(line l, point p) {
    return proj(p - l.r, l.v) + l.r;
}

point distVectorEndPoint(line l, point p) {
    return p;
}

point distVector(line a, line b) {
    point dr = b.r - a.r;
    point n = cross(a.v, b.v);
    if (n.norm() < EPS) {
        return dr - proj(dr, a.v);
    }
    else return proj(dr, n);
}

double dist(line a, line b) {
    return distVector(a, b).norm();
}

point distVectorBasePoint(line a, line b) {
    if (cross(a.v, b.v).norm() < EPS) return a.r;
    point d = distVector(a, b);
    double lambda;
    if (fabs(b.v.x*a.v.y - a.v.x*b.v.y) > EPS)
        lambda = (b.v.x*(b.r.y-a.r.y-d.y) - b.v.y*(b
            .r.x-a.r.x-d.x))/(b.v.x*a.v.y - a.v.x*b.
            v.y);
    else if (fabs(b.v.x*a.v.z - a.v.x*b.v.z) > EPS)
        lambda = (b.v.x*(b.r.z-a.r.z-d.z) - b.v.z*(b
            .r.x-a.r.x-d.x))/(b.v.x*a.v.z - a.v.x*b.
            v.z);
    else if (fabs(b.v.z*a.v.y - a.v.z*b.v.y) > EPS)
        lambda = (b.v.z*(b.r.y-a.r.y-d.y) - b.v.y*(b
            .r.z-a.r.z-d.z))/(b.v.z*a.v.y - a.v.z*b.
            v.y);
    return a.r + (a.v*lambda);
}

point distVectorEndPoint(line a, line b) {
    return distVectorBasePoint(a, b) + distVector(a,
        b);
}
```

8.17 Geometria Analítica

Pontos de intersecção de dois círculos:

$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} = dist(c_1, c_2) \tag{8.1}$$

$$l = \frac{r_1^2 - r_2^2 + d^2}{2d} \tag{8.2}$$

$$h = \sqrt{r_1^2 - l^2} \tag{8.3}$$

$$x = \frac{l}{d}(x_2 - x_1) \pm \frac{h}{d}(y_2 - y_1) + x_1 \tag{8.4}$$

$$y = \frac{l}{d}(y_2 - y_1) \mp \frac{h}{d}(x_2 - x_1) + y_1 \tag{8.5}$$

```
#include <algorithm>
#include <cmath>
using namespace std;

bool circleCircle(circle c1, circle c2, pair<point,
    point> & out) {
    double d = dist(c2.c, c1.c);
    double co = (d*d + c1.r*c1.r - c2.r*c2.r)/(2*d*c1
        .r);
    if (fabs(co) > 1.0) return false;
    double alpha = acos(co);
    point rad = (c2.c-c1.c)*(1.0/d*c1.r);
    out = {c1.c + rotate(rad, -alpha), c1.c + rotate(
        rad, alpha)};
    return true;
}
```

A equação da reta que passa pelos pontos (x_1, y_1) e (x_2, y_2) é dada por:

$$(y_2 - y_1)x + (x_1 - x_2)y + (y_1 x_2 - x_1 y_2) = 0 \tag{8.6}$$

Matrizes de rotação. Sentido positivo a partir da regra da mão direita e supondo $\vec{z} = \vec{x} \times \vec{y}$.

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos\theta & -sen\theta \\ 0 & sen\theta & cos\theta \end{bmatrix} \tag{8.7}$$

$$R_y(\theta) = \begin{bmatrix} cos\theta & 0 & sen\theta \\ 0 & 1 & 0 \\ -sen\theta & 0 & cos\theta \end{bmatrix} \tag{8.8}$$

$$R_z(\theta) = \begin{bmatrix} cos\theta & -sen\theta & 0 \\ sen\theta & cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{8.9}$$

Baricentro de um polígono de n lados. A = área com sinal. CUIDADO: Transladar polígono para a origem para não perder precisão senão toma WA! Depois transladar de volta.

$$A = \frac{1}{2}\sum_{i=0}^{n-1} p_i \times p_{i+1} \tag{8.10}$$

$$C_x = \frac{1}{6A}\sum_{i=0}^{n-1}(x_i + x_{i+1})(p_i \times p_{i+1}) \tag{8.11}$$

$$C_y = \frac{1}{6A}\sum_{i=0}^{n-1}(y_i + y_{i+1})(p_i \times p_{i+1}) \tag{8.12}$$

8.18 Coordenadas polares, cilíndricas e esféricas

Coordenadas polares:

$$x = rcos\phi \qquad y = rsen\phi \qquad dS = rdrd\phi \tag{8.13}$$

Coordenadas cilíndricas:

$$x = rcos\phi \qquad y = rsen\phi \qquad z = z \tag{8.14}$$

$$\vec{d\gamma} = dr\hat{r} + rd\phi\hat{\phi} + dz\hat{z} \qquad dV = rdrd\phi dz \tag{8.15}$$

$$d\vec{S}_r = rd\phi dz\hat{r} \qquad d\vec{S}_\phi = drdz\hat{\phi} \qquad d\vec{S}_z = rdrd\phi\hat{z} \tag{8.16}$$

$$\vec{\nabla} f = \frac{\partial f}{\partial r}\hat{r} + \frac{1}{r}\frac{\partial f}{\partial \phi}\hat{\theta} + \frac{\partial f}{\partial z}\hat{z} \tag{8.17}$$

$$\vec{\nabla}.\vec{F} = \frac{1}{r}\frac{\partial}{\partial r}(rF_r) + \frac{1}{r}\frac{\partial F_\phi}{\partial \phi} + \frac{\partial F_z}{\partial z} \tag{8.18}$$

$$\vec{\nabla} \times \vec{F} = (\frac{1}{r}\frac{\partial F_z}{\partial \phi} - \frac{\partial F_\phi}{\partial z})\hat{r} + (\frac{\partial F_r}{\partial z} - \frac{\partial F_z}{\partial r})\hat{\phi} + \frac{1}{r}(\frac{\partial}{\partial r}(rF_\phi) - \frac{\partial F_r}{\partial \phi})\hat{z} \tag{8.19}$$

Coordenadas esféricas:

$$x = rcos\phi sen\theta \qquad y = rsen\phi sen\theta \qquad z = rcos\theta \tag{8.20}$$

$$\vec{d\gamma} = dr\hat{r} + rd\theta\hat{\theta} + rsen\theta d\phi\hat{\phi} \tag{8.21}$$

$$d\vec{S}_r = r^2 sen\theta d\theta d\phi\hat{r} \quad d\vec{S}_\theta = rsen\theta d\phi dr\hat{\theta} \quad d\vec{S}_\phi = rdrd\theta\hat{\phi} \tag{8.22}$$

$$dV = r^2 sen\theta drd\theta d\phi \qquad d\Omega = \frac{dS_r}{r^2} = sen\theta d\theta d\phi \tag{8.23}$$

$$\vec{\nabla} f = \frac{\partial f}{\partial r}\hat{r} + \frac{1}{r}\frac{\partial f}{\partial \theta}\hat{\theta} + \frac{1}{rsen\theta}\frac{\partial f}{\partial \phi}\hat{\phi} \tag{8.24}$$

$$\vec{\nabla}.\vec{F} = \frac{1}{r^2}\frac{\partial}{\partial r}(r^2F_r) + \frac{1}{rsen\theta}\frac{\partial}{\partial \theta}(sen\theta F_\theta) + \frac{1}{rsen\theta}\frac{\partial F_\phi}{\partial \phi} \tag{8.25}$$

$$\vec{\nabla} \times \vec{F} = \frac{1}{rsen\theta}(\frac{\partial}{\partial \theta}(F_\phi sen\theta) - \frac{\partial F_\theta}{\partial \phi})\hat{r} + \frac{1}{r}(\frac{1}{sen\theta}\frac{\partial F_r}{\partial \phi} - \frac{\partial}{\partial r}(rF_\phi))\hat{\theta} + \frac{1}{r}(\frac{\partial}{\partial r}(rF_\theta) - \frac{\partial F_r}{\partial \theta})\hat{\phi} \tag{8.26}$$

8.19 Cálculo Vetorial 2D

Curva regular no plano e comprimento do arco:

$$\vec{\gamma}(t),\ C^1,\ \vec{\gamma}'(t) \neq 0 \qquad L(\gamma) = \int_a^b ||\vec{\gamma}'(t)||dt \tag{8.27}$$

Reta tangente e normal:

$$T : X = \vec{\gamma}(t_0) + \lambda.\vec{\gamma}'(t_0) \tag{8.28}$$

$$N : \{X \in R^2 :< X - \vec{\gamma}(t_0), \vec{\gamma}'(t_0) >= 0\} \tag{8.29}$$

Curva de orientação invertida:

$$\vec{\gamma}^-(t) = \vec{\gamma}(a + b - t) \tag{8.30}$$

Referencial de Frenet:

$$\vec{T}(t) = \frac{\vec{\gamma}'(t)}{||\vec{\gamma}'(t)||} \qquad \vec{N}(t) = (-T_y(t), T_x(t)) \tag{8.31}$$

Curvatura, raio e centro de curvatura:

$$K(t) = \frac{\vec{\gamma}''(t).\vec{N}(t)}{||\vec{\gamma}'(t)||^2} \tag{8.32}$$

$$R(t) = \frac{1}{|k(t)|} \qquad \vec{C}(t) = \vec{\gamma}(t) + \frac{\vec{N}(t)}{K(t)} \tag{8.33}$$

Equações de Frenet:

$$\vec{T}'(t) = K(s).\vec{N}(t) \qquad \vec{N}'(t) = -K(t).\vec{T}(t) \tag{8.34}$$

Teorema de Gauss no plano:

$$\int_{\partial S} \vec{F}.\vec{N}d\gamma = \int\int_S \vec{\nabla}.\vec{F}dxdy \tag{8.35}$$

Teorema de Green:

$$\int_{\partial\Omega} Pdx + Qdy = \int\int_\Omega (\frac{\partial Q}{\partial x} - \frac{\partial P}{\partial y})dxdy \tag{8.36}$$

8.20 Cálculo Vetorial 3D

Referencial de Frenet:

$$\vec{T}(t) = \frac{\vec{\gamma}'(t)}{||\vec{\gamma}'(t)||} \quad \vec{B}(t) = \frac{\vec{\gamma}'(t) \times \vec{\gamma}''(t)}{||\vec{\gamma}'(t) \times \vec{\gamma}''(t)||} \quad \vec{N}(t) = \vec{B}(t) \times \vec{T}(t) \tag{8.37}$$

Curvatura e torção:

$$\tau(t) = \frac{< \vec{\gamma}'(t) \times \vec{\gamma}''(t), \vec{\gamma}'''(t) >}{||\vec{\gamma}'(t) \times \vec{\gamma}''(t)||^2} \quad K(t) = \frac{||\vec{\gamma}'(t) \times \vec{\gamma}''(t)||}{||\vec{\gamma}'(t)||^3} \tag{8.38}$$

Plano normal a $\vec{\gamma}(t_0)$:

$$< X - \vec{\gamma}(t_0), T(t_0) >= 0 \tag{8.39}$$

Equações de Frenet:

$$\vec{T}'(t) = K(t).\vec{N}(t) \tag{8.40}$$

$$\vec{N}'(t) = -K(t).\vec{T}(t) - \tau(t).\vec{B}(t) \tag{8.41}$$

$$\vec{B}'(t) = -\tau(t).\vec{N}(t) \tag{8.42}$$

Integral de linha de um campo escalar:

$$\int_\gamma f d\gamma = \int_a^b f(\vec{\gamma}(t))||\vec{\gamma}'(t)||dt \tag{8.43}$$

Integral de linha de um campo vetorial:

$$\int_\gamma \vec{F}.d\vec{\gamma} = \int_a^b \vec{f}(\vec{\gamma}(t)).\vec{\gamma}'(t)dt \tag{8.44}$$

Operador nabla:

$$\vec{\nabla} = (\frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z}) \tag{8.45}$$

Campo gradiente:

$$\vec{F}(x,y,z) = \vec{\nabla} f(x,y,z) = (\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z})(x,y,z) \tag{8.46}$$

Campo conservativo:

$$\vec{F}(x,y,z) = \vec{\nabla} f(x,y,z) \Leftrightarrow \int_\gamma \vec{F}.d\vec{\gamma} = 0 \tag{8.47}$$

Campo rotacional:

$$\vec{\nabla} \times \vec{F} = (\frac{\partial R}{\partial y} - \frac{\partial Q}{\partial z}, \frac{\partial P}{\partial z} - \frac{\partial R}{\partial x}, \frac{\partial Q}{\partial x} - \frac{\partial P}{\partial y}) \tag{8.48}$$

$\vec{F}$ é conservativo $\rightarrow \vec{\nabla} \times \vec{F} = 0$ Campo divergente:

$$\vec{\nabla}.\vec{F} = (\frac{\partial P}{\partial x}, \frac{\partial Q}{\partial y}, \frac{\partial R}{\partial z}) \tag{8.49}$$

- $\vec{F}$ é solenoidal quando $\vec{\nabla}.\vec{F} = 0$
- Existe $\vec{G}$ tal que $\vec{F} = \vec{\nabla} \times \vec{G} \Rightarrow \vec{\nabla}.\vec{F} = 0$
- $\vec{\nabla}.\vec{F} \neq 0 \Rightarrow$ não existe $\vec{G}$ tal que $\vec{F} = \vec{\nabla} \times \vec{G}$

Superfície parametrizada:

$$\vec{S}(u,v) = (x(u,v), y(u,v), z(u,v)) \tag{8.50}$$

$$\vec{S_u}(u,v) = (\frac{\partial x}{\partial u}, \frac{\partial y}{\partial u}, \frac{\partial z}{\partial u}), \ \vec{S_v}(u,v) = (\frac{\partial x}{\partial v}, \frac{\partial y}{\partial v}, \frac{\partial z}{\partial v}) \tag{8.51}$$

Vetor normal à superfície:

$$\vec{N}(u,v) = (\vec{S_u} \times \vec{S_v})(u,v) \neq \vec{0} \tag{8.52}$$

Superfície diferenciável:

$$\vec{N}(u,v) \neq \vec{0} \tag{8.53}$$

Plano tangente à superfície:

$$< (x,y,z) - \vec{S}(u_0,v_0), \vec{N}(u_0,v_0) >= 0 \tag{8.54}$$

Área da superfície:

$$A(S) = \int\int_U ||\vec{N}(u,v)||dudv \tag{8.55}$$

Integral de superfície de um campo escalar:

$$\int\int_S f dS = \int\int_U f(\vec{S}(u,v))||\vec{N}(u,v)||dudv \tag{8.56}$$

Integral de superfície de um campo vetorial:

$$\int\int_S \vec{F}.d\vec{S} = \int\int_U \vec{F}(\vec{S}(u,v)).\vec{N}(u,v)dudv \tag{8.57}$$

Massa e centro de massa:

$$M = \int_\gamma \rho(t) d\vec{\gamma}, \qquad \overline{x}M = \int_\gamma \vec{x}(t)\rho(t)d\vec{\gamma} \tag{8.58}$$

$$M = \int\int_S \rho(u,v)ds \quad \overline{x}M = \int\int_S \vec{x}(u,v)\rho(u,v)ds \tag{8.59}$$

$$M = \int\int\int_V \rho(x,y,z)dv \quad \overline{x}M = \int\int_S \vec{x}(x,y,z)\rho(z,y,z)dv \tag{8.60}$$

Teorema de Pappus para a área, d = distância entre $\overline{x}$ e o eixo de rotação:

$$A(S) = 2\pi d L(C) \tag{8.61}$$

Teorema de Pappus para o volume, d = distância entre $\overline{x}$ e o eixo de rotação:

$$V(\Omega) = 2\pi d A(S) \tag{8.62}$$

Teorema de Gauss no espaço:

$$\int\int_{\partial V} \vec{F}.\vec{N}d\gamma = \int\int\int_V \vec{\nabla}.\vec{F}dxdydz \tag{8.63}$$

Teorema de Stokes:

$$\int_{\partial S} \vec{F}.d\vec{\gamma} = \int\int_S \vec{\nabla} \times \vec{F}.\vec{N}ds \tag{8.64}$$

8.21 Problemas de precisão, soma estável e fórmula de bháskara

Para IEEE 754:

- Números são representados pelo seu valor representável mais próximo.
- Operações básicas $(+, -, *, /, \sqrt{})$ são feitas com precisão infinita e depois arredondadas.

tipo	bits	expoente	precisão binária	precisão decimal	ϵ
float	32	38	24	6	$2^{-24} \approx 5.96 \times 10^{-8}$
double	64	308	53	15	$2^{-53} \approx 1.11 \times 10^{-16}$
long double	80	19.728	64	18	$2^{-64} \approx 5.42 \times 10^{-20}$

Todas as operações somente com *int* podem ser feitas perfeitamente com *double*. Com *long long*, *long double*.

Sempre que possível, trabalhar com erro absoluto, e não relativo. O erro relativo pode disparar no caso de cancelamento catastrófico - subtração de dois números com mesma ordem de grandeza. Seja ϵ o erro relativo natural do tipo e M o valor máximo de uma variável. O erro absoluto é dado por $M\epsilon$. Para as operações entre números:

- Soma e subtração normal: O erro absoluto após n somas é $nM\epsilon$. Cada soma ou subtração de não inteiro adiciona erro relativo ϵ. O erro relativo da soma de um número obtido com k_1 somas com um de k_2 somas é $(k_1 + k_2 + 1)\epsilon$.
- Soma de números não negativos: O erro relativo da soma de um número obtido com k_1 somas de números não negativos com um de k_2 somas é $(max(k_1, k_2) + 1)\epsilon$. Usando um esquema de árvore (ou divisão e conquista) para realizar a soma de n elementos não negativos, pode-se obter erro de $2M\epsilon \log_2 n$. Ver struct *StableSum* abaixo.
- Multiplicação: Seja d a dimensão da resposta da operação (número máximo de multiplicações em série). O erro absoluto de combinações de n operações $(+, -, *)$ é $M^d((1 + \epsilon)^n - 1) \approx nM^d\epsilon$, se $n\epsilon << 1$.
- Radiciação: Uma imprecisão δ se torna uma imprecisão $\sqrt{\delta}$. Ou seja, um erro absoluto de $nM^d\epsilon$ se torna $\sqrt{nM^d\epsilon}$. Cuidado, por exemplo, se $\delta = 10^{-6}, \sqrt{\delta} = 10^{-3}$.
- Divisão: Não é possível estimar o erro da divisão. Se um número x possui a mesma ordem de grandeza que seu erro absoluto, então $1/x$ pode assumir qualquer valor positivo ou negativo ou não existir. Evitar fazer divisão por variáveis.

A struct *StableSum* (soma estável) executa a soma de n números não negativos com erro absoluto $2M\epsilon \log_2 n$. *val*() retorna a soma total. Semelhante à Fenwick Tree, funciona em $O(1)$ amortizado na forma estática. Caso os números sejam dinâmicos, usar a Fenwick Tree mesmo.

A função *quadRoots* retorna quantas raízes uma equação quadrática tem e as bota em *out*. Ela evita o cancelamento catastrófico causado quando $|b| \approx \sqrt{b^2 - 4ac}$, ou seja, $ac \approx 0$, usando ambas as fórmulas de bháskara:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} = \frac{2c}{-b \mp \sqrt{b^2 - 4ac}} \tag{8.65}$$

```
#include <vector>
#include <cmath>
using namespace std;

struct StableSum {
    int cnt = 0;
    vector<double> v, pref(1, 0);
    void operator += (double a) { // a >= 0
        for (int s = ++cnt; s % 2 == 0; s >>= 1) {
            a += v.back();
            v.pop_back(), pref.pop_back();
        }
        v.push_back(a);
        pref.push_back(pref.back() + a);
    }
    double val() { return pref.back(); }
};

int quadRoots(double a, double b, double c, pair<
    double, double> &out) {
    if (fabs(a) < EPS) return 0;
    double delta = b*b - 4*a*c;
    if (delta < 0) return 0;
    double sum = (b >= 0) ? -b-sqrt(delta) : -b+sqrt(
        delta);
    out = {sum/(2*a), fabs(sum) > EPS ? 0 : (2*c)/sum
        };
    return 1 + (delta > 0);
}
```

Capítulo 9

Miscelânea

9.1 Algoritmo de Mo

Resolve queries offline em $O((N+Q)\sqrt{N})$. É necessário que o update de queries seja $O(1)$. Pra usar na árvore em query de subárvore, escrever na forma pre-ordem. Cada subárvore é um trecho contíguo na array. Com query de caminho, escrever a árvore na forma prepos-ordem. Se um elemento aparece duas vezes no intervalo, ele NÃO está no caminho. o LCA dos dois elementos deve ser tratado separadamente nos casos da informação estar no nó.

```
#define SQ 500
int curans; // current ans
void add(int v) { /* add value */ }
void rem(int v) { /* remove value */ }

struct query {
   int i, l, r, ans;
};

bool qcomp(query a, query b) {
   return a.l/SQ == b.l/SQ ? a.r < b.r : a.l < b.l;
}
bool icomp(query a, query b) { return a.i < b.i; }
```

```
void mo(int v[], query qs[], int Q) {
   int l = 0, r = -1;
   sort(qs, qs + Q, qcomp);
   for (int i = 0; i < Q; i++) {
      query & q = qs[i];
      while (r < q.r) add(v[++r]);
      while (r > q.r) rem(v[r--]);
      while (l < q.l) rem(v[l++]);
      while (l > q.l) add(v[--l]);
      q.ans = curans;
   }
   sort(qs, qs + Q, icomp);
}
```

9.2 Calendário gregoriano

count retorna a diferença de dias entre a data e 01/01/0001. *weekday* retorna o dia da semana indexado em 0, iniciando no domingo. *leap* retorna se o ano é bissexto. *advance* anda a data em tantos dias. Todas as funções $O(1)$.

```
int mnt[13] = {0, 31, 59, 90, 120, 151, 181, 212,
    243, 273, 304, 334, 365};

struct Date {
   int d, m, y;
   Date() : d(1), m(1), y(1) {}
   Date(int d, int m, int y) : d(d), m(m), y(y) {}
   Date(int days) : d(1), m(1), y(1) { advance(days)
       ; }
   bool leap() { return (y%4 == 0 && y%100) || (y
       %400 == 0); }
   int mdays() { return mnt[m] - mnt[m-1] + (m == 2)
       *leap(); }
   int ydays() { return 365 + leap(); }
   int count() { // dist to 01/01/01
      return (d-1) + mnt[m-1] + (m > 2)*leap() +
         365*(y-1) + (y-1)/4 - (y-1)/100 + (y-1)
             /400;
   }
   int weekday() { return (count()+1) % 7; }
   void advance(int days) {
      days += count();
      d = m = 1, y = 1 + days/366;
      days -= count();
      while(days >= ydays()) days -= ydays(), y++;
      while(days >= mdays()) days -= mdays(), m++;
      d += days;
   }
};
```

9.3 Simplex

Algoritmo para resolver problemas de programação linear em que se deseja maximizar/minimizar um custo de N variáveis, dado por $\sum_{i=1}^{n} a_i x_i$, respeitando M condições de restrição ($\geq$, $\leq$, $=$). Tem complexidade esperada $O(N^2M)$ e pior caso

exponencial. Para utilizar este algoritmo, chame o método *init* com a função objetivo, *add constraint* para adicionar restrições e *solve* para computar a solução ótima.

```
#define MAXN 109
#define MAXM 10009
#define EPS 1e-9

#define MINIMIZE -1
#define MAXIMIZE +1
#define LESSEQ -1
#define EQUAL 0
#define GREATEQ 1
#define INFEASIBLE -1
#define UNBOUNDED 999

/***
1. m is the number of constraints indexed from 1 to
    m, and n is the number of variables indexed from
     0 to n-1
2. ar[0] contains the objective function f, and ar
    [1] to ar[m] contains the constraints, ar[i][n]
    = lim_i
3. All xi >= 0
4. If xi <= 0, replace xi by r1 - r2 (Number of
    variables increases by one, -x, +r1, +r2)
5. solution_flag = INFEASIBLE if no solution is
    possible and UNBOUNDED if no finite solution is
    possible
6. After successful completion, val[] contains the
    values of x0, x1 .... xn for the optimal value
    returned
7. If ABS(X) <= M in constraints, Replace with X <=
    M and -X <= M
8. Fractional LP:

    max/min
        3x1 + 2x2 + 4x3 + 6
        ---------------------
        3x1 + 3x2 + 2x3 + 5

        s.t. 2x1 + 3x2 + 5x3 >= 23
        x1, x2, x3 >= 0

    Replace with:

    max/min
        3y1 + 2y2 + 4y3 + 6t

        s.t. 3y1 + 3y2 + 2y3 + 5t = 1
        2y1 + 3y2 + 53 - 23t >= 0
        y1, y2, y3, t >= 0
***/

namespace lp {
double val[MAXN], ar[MAXM][MAXN];
int m, n, solution_flag, minmax_flag, basis[MAXM],
    index[MAXN];

void init(int nvars, double* f, int flag) {
   solution_flag = 0;
   ar[0][nvars] = 0.0;
   m = 0, n = nvars, minmax_flag = flag;
   for (int i = 0; i < n; i++)
      ar[0][i] = f[i] * minmax_flag;
}

void add_constraint(double* C, double lim, int cmp)
    {
   m++, cmp *= -1;
   if (cmp == EQUAL) {
      for (int i = 0; i < n; i++) ar[m][i] = C[i];
      ar[m++][n] = lim;
      for (int i = 0; i < n; i++) ar[m][i] = -C[i];
      ar[m][n] = -lim;
   }
   else {
      for (int i = 0; i < n; i++) ar[m][i] = C[i] *
          cmp;
      ar[m][n] = lim * cmp;
   }
}

void init() {
   for (int i = 0; i <= m; i++) basis[i] = -i;
   for (int j = 0; j <= n; j++)
      ar[0][j] = -ar[0][j], index[j] = j, val[j] =
          0;
}

inline void pivot(int m, int n, int a, int b) {
   for (int i = 0; i <= m; i++)
      for (int j = 0; i != a && j <= n; j++)
         if (j != b) ar[i][j] -= (ar[i][b] * ar[a][j
             ]) / ar[a][b];
   for (int j = 0; j <= n; j++)
      if (j != b) ar[a][j] /= ar[a][b];
   for (int i = 0; i <= m; i++)
      if (i != a) ar[i][b] = -ar[i][b] / ar[a][b];
   ar[a][b] = 1.0 / ar[a][b];
   swap(basis[a], index[b]);
}

inline double solve() {
   init();
   int i, j, k, l;
   while (true) {
      for (i = 1, k = 1; i <= m; i++)
         if ((ar[i][n] < ar[k][n]) || (ar[i][n] ==
             ar[k][n] && basis[i] < basis[k] && (
             rand() & 1))) k = i;
      if (ar[k][n] >= -EPS) break;
      for (j = 0, l = 0; j < n; j++)
         if ((ar[k][j] < (ar[k][l] - EPS)) || (ar[k
             ][j] < (ar[k][l] - EPS) && index[i] <
             index[j] && (rand() & 1))) l = j;
      if (ar[k][l] >= -EPS) {
         solution_flag = INFEASIBLE;
         return -1.0;
      }
      pivot(m, n, k, l);
   }
   while (true) {
      for (j = 0, l = 0; j < n; j++)
         if ((ar[0][j] < ar[0][l]) || (ar[0][j] ==
             ar[0][l] && index[j] < index[l] && (
             rand() & 1))) l = j;
      if (ar[0][l] > -EPS) break;
      for (i = 1, k = 0; i <= m; i++)
         if (ar[i][l] > EPS && (!k || ar[i][n] / ar[
             i][l] < ar[k][n] / ar[k][l] - EPS || (
             ar[i][n] / ar[i][l] < ar[k][n] / ar[k][
             l] + EPS && basis[i] < basis[k]))) k =
             i;
      if (ar[k][l] <= EPS) {
         solution_flag = UNBOUNDED;
         return -999.0;
      }
      pivot(m, n, k, l);
   }
   for (i = 1; i <= m; i++)
      if (basis[i] >= 0) val[basis[i]] = ar[i][n];
```

```
    solution_flag = 1;
    return (ar[0][n] * minmax_flag);
}
}
```

9.4 Iteração sobre polyominos

Itera sobre todas as possíveis figuras de polyominos em uma grade. Posições com $num[i][j] \neq 0$ são proibidas. Neste algoritmo em específico, calcula a soma máxima. Caso precise da forma, guardar a pilha de recursão. Número do polyominos:

tamanho	1	2	3	4	5	6	7	8	9	10	11	12
quantidade	1	2	6	19	63	216	760	2,725	9,910	36,446	135,268	505,861

```
#define MAXN 59
int N, M;
int num[MAXN][MAXN], qi[MAXN*MAXN], qj[MAXN*MAXN];
int field[MAXN][MAXN], par[MAXN][MAXN];
int di[4] = {0, 1, 0, -1};
int dj[4] = {-1, 0, 1, 0};
int cnt;

void assign(int i, int j, int k) {
    qi[k] = i; qj[k] = j; num[i][j] = k;
}

#define valid(i, j) (i >= 0 && i < N && j >= 0 && j
    < M)

int iterate(int k, int h) {
    if (h == 0) return 0;
    int i = qi[k], j = qj[k], ni, nj;
    for(int d = 0; d < 4; d++) {
        ni = i + di[d]; nj = j + dj[d];
        if (!valid(ni, nj)) continue;
        if (num[ni][nj] == 0) {
            par[ni][nj] = k;
            assign(ni, nj, ++cnt);
        }
    }
    int ans = 0, cur;
    for(int t = k+1; t <= cnt; t++) {
        cur = iterate(t, h-1);
        if (cur > ans) ans = cur;
    }
    for(int d = 0; d < 4; d++) {
        ni = i + di[d]; nj = j + dj[d];
        if (!valid(ni, nj)) continue;
        if (par[ni][nj] == k) {
            par[ni][nj] = 0; cnt--;
            assign(ni, nj, 0);
        }
    }
    return ans + field[i][j];
}
```

9.5 Quadrado Mágico Ímpar

Gera uma matriz quadrática $n \times n$ em $O(n^2)$, n ímpar, tal que a soma dos elementos ao longo de todas as linhas, de todas as colunas e das duas diagonais é a mesma. Os elementos vão de 1 a n^2. A matriz é indexada em 0.

```
#define MAXN 1009
int mat[MAXN][MAXN], n; //0-indexed

void magicsquare() {
   int i=n-1, j=n/2;
   memset(&mat, 0, sizeof mat);
   for(int k = 1; k <= n*n; k++) {
      mat[i][j] = k;
      if (mat[(i+1)%n][(j+1)%n] > 0) {
         i = (i-1+n)%n;
      }
      else {
         i = (i+1)%n;
         j = (j+1)%n;
      }
   }
}
```

9.6 Ciclos em sequências: Algoritmo de Floyd

```
ii floydCycleFinding(int x0) {
   // 1st part: finding k*start, hares speed is 2x
       tortoises
   int tortoise = f(x0), hare = f(f(x0)); // f(x0)
       is the node next to x0
   while (tortoise != hare) { tortoise = f(tortoise)
       ; hare = f(f(hare)); }
   // 2nd part: finding start, hare and tortoise
       move at the same speed
   int start = 0; hare = x0;
   while (tortoise != hare) { tortoise = f(tortoise)
       ; hare = f(hare); start++;}
   // 3rd part: finding period, hare moves, tortoise
        stays
   int period = 1; hare = f(tortoise);
   while (tortoise != hare) { hare = f(hare); period
       ++; }
   return ii(start, period);
}
```

9.7 Expressão Parentética para Polonesa

Dada uma expressão matemática na forma parentética, converte para a forma polonesa e retorna o tamanho da string na forma polonesa. Ex.: $(2 * 4/a \wedge b)/(2 * c) \rightarrow 24 * ab \wedge /2c * /$.

```
inline bool isOp(char c) {
   return c=='+' || c=='-' || c=='*' || c=='/' || c
       =='^';
}

inline bool isCarac(char c) {
   return (c>='a' && c<='z') || (c>='A' && c<='Z')
       || (c>='0' && c<='9');
}

int paren2polish(char* paren, char* polish) {
   map<char, int> prec;
   prec['('] = 0;
   prec['+'] = prec['-'] = 1;
   prec['*'] = prec['/'] = 2;
   prec['^'] = 3;
   int len = 0;
   stack<char> op;
   for (int i = 0; paren[i]; i++) {
      if (isOp(paren[i])) {
         while (!op.empty() && prec[op.top()] >=
             prec[paren[i]]) {
            polish[len++] = op.top(); op.pop();
         }
         op.push(paren[i]);
      }
      else if (paren[i]=='(') op.push('(');
      else if (paren[i]==')') {
         for (; op.top()!='('; op.pop())
            polish[len++] = op.top();
         op.pop();
      }
      else if (isCarac(paren[i]))
         polish[len++] = paren[i];
   }
   for(; !op.empty(); op.pop())
      polish[len++] = op.top();
   polish[len] = 0;
   return len;
}
```

9.8 Problema de Josephus

Em um círculo de n jogadores (0-indexed) em que eles se matam a cada k em ordem crescente. O índice do sobrevivente pode ser calculado pela recursão: $f(n,k) = (f(n-1,k)+k)\%n, f(1,k) = 0$.

9.9 Problema do histograma

Algoritmo $O(n)$ para resolver o problema do retângulo máximo em um histograma.

```
#define MAXN 100009

ll histogram(ll vet[], int n) {
    stack<ll> s;
    ll ans = 0, tp, cur;
    int i = 0;
    while(i < n || !s.empty()) {
        if (i < n && (s.empty() || vet[s.top()] <= vet
            [i])) s.push(i++);
        else {
            tp = s.top();
            s.pop();
            cur = vet[tp] * (s.empty() ? i : i - s.top
                () - 1);
            if (ans < cur) ans = cur;
        }
    }
    return ans;
}
```

9.10 Problema do casamento estável

Resolve o problema do casamento estável em $O(n^2)$. m é o número de homens, n é o número de mulheres, $L[i]$ é a lista de preferências do i-ésimo homem (melhor primeiro), $R[i][j]$ é a nota que a mulher i dá ao homem j. $R2L[i] == j$ retorna se a mulher i está casada com o homem j, $L2R[i] == j$ retorna se o homem i está casado com a mulher j.

```
#define MAXN 1009
int m, n, p[MAXN];
int L[MAXN][MAXN], R[MAXN][MAXN];
int R2L[MAXN], L2R[MAXN];

void stableMarriage() {
    memset(R2L, -1, sizeof(R2L));
    memset(p, 0, sizeof(p));
    for(int i = 0, wom, hubby; i < m; i++) {
        for (int man = i; man >= 0;) {
            while(true) {
                wom = L[man][p[man]++];
                if( R2L[wom] < 0 || R[wom][man] > R[
                    wom][R2L[wom]] ) break;
            }
            hubby = R2L[wom];
            R2L[L2R[man] = wom] = man;
            man = hubby;
        }
    }
}
```

9.11 Código de Huffman

Computa o custo do autômato de Huffman: dado um conjunto de elementos, montar uma árvore binária cujas folhas são os elementos minimizando: $cost = \sum_{i=0}^{n-1} a[i] \times depth[i]$.

```
ll huffman(ll* a, int n) {
    ll ans = 0, u, v;
    priority_queue<ll> pq;
    for(int i=0; i<n; i++) pq.push(-a[i]);
    while(pq.size() > 1) {
        u = -pq.top(); pq.pop();
        v = -pq.top(); pq.pop();
        pq.push(-u-v);
        ans += u + v;
    }
    return ans;
}
```

9.12 Problema do Cavalo

Como achar, em um tabuleiro, um caminho hamiltoniano com o cavalo? Backtrack usando como heurística a regra de Warsndorf: visite primeiro lugares com o menor número de saídas.

9.13 Intersecção de Matróides

Encontra o conjunto independente máximo na intersecção de dois matróides. Um exemplo é a maior Spanning Tree com arestas de cores diferentes. Complexidade até $O(N * M^2)$, na prática bem menor. O algoritmo procura sequências aumentantes, fazendo um grafo bipartido entre arestas dentro/fora do conjunto máximo, e BFS de Q (arestas fora de cores não usadas) a T (arestas fora que não formam ciclo).

```
#define MAXM 1009
vector<set<ii>> adjList;
vector<vector<int>> colors;
vector<bool> T, Q, inSet, usedColor;
ii edges[MAXM];
int c[MAXM], N, M, C;

class UnionFind { ... };

void findCycle(int s, int t, vector<int>& cycle) {
   vector<ii> prv;
   queue<int> q;
   prv.assign(N, ii(-1, -1));
   q.push(s); prv[s] = ii(s, -1);
   while (!q.empty()) {
      int u = q.front(); q.pop();
      if (u == t) break;
      for (set<ii>::iterator it = adjList[u].begin()
          ; it != adjList[u].end(); it++) {
         int v = (*it).first;
         if (prv[v].first == -1) {
            prv[v] = ii(u, (*it).second);
            q.push(v);
         }
      }
   }
   while (t != s) {
      cycle.push_back(prv[t].second);
      t = prv[t].first;
   }
}

void greedyAugmentation(UnionFind &UF) {
   int e, u, v;
   for (int i = 0; i < M; i++) {
      u = edges[i].first, v = edges[i].second;
      if (!UF.isSameSet(u, v) && !usedColor[c[i]]) {
         UF.unionSet(u, v);
         adjList[u].insert(ii(v, i));
         adjList[v].insert(ii(u, i));
         usedColor[c[i]] = true;
         inSet[i] = true;
      }
   }
}

int bfs(vector<int>& prv) {
   queue<int> q;
   prv.assign(M, -1);
   for (int i = 0; i < M; i++) {
      if (Q[i]) { q.push(i); prv[i] = i; }
   }
   int u;
   while (!q.empty()) {
      u = q.front(); q.pop();
      if (T[u]) return u;
      // If in independent set, check others from
          same color
      if (inSet[u]) {
         for (int i = 0; i < colors[c[u]].size(); i
             ++) {
            int v = colors[c[u]][i];
            if (v != u && prv[v] == -1) {
               prv[v] = u; q.push(v);
            }
         }
      } else { // If not independent, check cycles
         vector<int> cycle;
         findCycle(edges[u].first, edges[u].second,
             cycle);
         for (int i = 0; i < cycle.size(); i++) {
            if (prv[cycle[i]] == -1) {
               prv[cycle[i]] = u; q.push(cycle[i]);
            }
         }
      }
   }
   return -1;
}

bool augment(UnionFind& UF) {
   Q.assign(M, false); T.assign(M, false);
   for (int i = 0; i < M; i++) {
      if (!UF.isSameSet(edges[i].first, edges[i].
          second)) { T[i] = true; }
      if (!usedColor[c[i]]) { Q[i] = true; }
   }

   vector<int> prv;
   int u = bfs(prv);
   if (u == -1) return false;
   UF.unionSet(edges[u].first, edges[u].second);
   while (true) {
      if(inSet[u]) {
         adjList[edges[u].first].erase(ii(edges[u].
             second, u));
         adjList[edges[u].second].erase(ii(edges[u].
             first, u));
      } else {
         adjList[edges[u].first].insert(ii(edges[u].
             second, u));
         adjList[edges[u].second].insert(ii(edges[u
             ].first, u));
      }

      inSet[u] = !inSet[u];
      if (u == prv[u]) break;
      u = prv[u];
   }
   usedColor[c[u]] = true;
   return true;
}

int maxIndependentSet() {
    inSet.assign(M, false); usedColor.assign(C,
        false);
    adjList.clear(); adjList.resize(N+5);

   UnionFind UF(N);
   greedyAugmentation(UF);
   while (augment(UF));

   int sz = 0;
   for (int i = 0; i < M; i++) if(inSet[i]) sz++;
   return sz;
}
```