Feuille de TD N^o 4 : Tableaux et quelques Tris

1 Hachage

Un hétérogramme est un mot ne contenant pas deux fois la même lettre. Quels sont les plus longs que vous trouvez ?

- But de l'exo : faire constater que ce n'est pas si facile et qu'il ne s'agit pas par exemple de faire un mot sans deux "e", mais qu'il y a facilement une double lettre qui traîne des qu'un mot est long, cette lettre étant variable d'un mot a l'autre. C'est un effet "conflit", similaire au "paradoxe des anniversaires" (il suffit de runir 23 personnes pour que la proba qu'il y en ait deux avec la même date d'anniversaire). Difficile de quantifier la proba d'avoir une double lettre dans un mot long, car la proba d'apparition d'une lettre est variable, dépend des lettres environnantes, etc. mais on constate bien l'apparition du phénomène. Trouvez-vous un hétérogramme de longueur 8 ? 9 ? plus ?

  Allez, il y en un joli de longueur 11 : algorithmes, cf aussi palindromes

  Le record es de 14 : stylographique (relatif au stylographe, ie au stylo, porte plume a reservoir d'encre), cryptogamiques (relatif a la cryptogamie, caractere des plantes cryptogames, ie dont les organes de la fructification sont peu apparents ou cachs, tels que les mousses, les fougres, les lichens, etc.) et xylographiques (relatif a la xylographie, Art dimprimer avec des caractres de bois, ou avec des planchettes de bois dans lesquelles sont tailles les lettres.)

  Les trois precedents sont donnes par wikipedia, mais erratum addendum, en fait, il y en a d'autres :

  brachylogiques (relatif a la maniere de s'exprimer par sentences), , hydnocarpiques (acide h. C16H28O2), divulgacherons/ont, en/deschtroumpfai, lycanthropique, dyschromatique (d'une mauvaise couleur, ou qui altere la couleur), macrophytiques (relatif au végétaux aquatiques visibles a l'oeil nu), kymographiques (relatif a l'enregistrement graphique des mouvements par exemple d'un organe (medical))

  et le record est en fait de 15 : lycanthropiques, relatifs au lycanthrope, loup-garou, ou malade se prenant pour un loup.

  Quelques autres : ,

  Mots de douze lettres : introuvables, indomptables, incomptables, monarchiques.

  Mots de onze lettres : parchemins, rudoyaient, confitures, comprenait, couvraient, courtisane, expliquant, incomplets, redoublait, importance, compensait, absolument, decouvrait, champenois, decouvrant, harmonieux, expliquons, hypocrites, debouchant, diplomates, spoliateur, monarchies, debauchons, boulevards, embarquons, compagnies, souviendra, amphitryon, formalites, conservait, monticules, imprudents, comblaient, prodiguant, gouvernait, encombrait, longitudes, francisque, calembours, galipoteux, pantoufles, brodequins, plaiderons, produisant, introuvable.

  En anglais, 16 lettres UNCOPYRIGHTABLES (avec s pluriel si on le prend comme un nom), 15 lettres HYDROPNEUMATICS

Rangez les lettres de a à k dans un tableau T[1..13].
Le faire avec le hachage avec listes chaînées,
le hachage linéaire, le double hachage.
On suppose que h et $h2$ de ces lettres sont :

lettre	*a*	*b*	*c*	*d*	*e*	*f*	*g*	*h*	*i*	*j*	*k*
$h(lettre)$	13	8	3	8	4	13	2	2	5	9	1
$h2(lettre)$	5	6	11	4	3	12	1	9	4	7	3

- listes chainees : T[13] contiendra un pointeur vers une liste chainee contenant a et f, T[12] un pointeur null, etc.

  hachage lineaire :

  f1 g2 c3 e4 h5 i6 k7 b8 d9 j10 a13

  a va en 13, b en 8, c en 3, d la place 8 est prise donc va en 9, e en 4, f place 13 prise donc va en "14 mod 13" donc en 1, g va en 2, h place 2 prise puis place 3 prise puis 4 aussi donc va en 5, i place 5 prise donc va en 6, j place 9 prise donc va en 10, k en 7

  Remarquez comment de gros paquets, "des grumeaux" se forment. Plus un grumeau est gros, plus on risque de tomber dedans, ce qui a pour effet de le faire encore plus grossir, ce qui empirer quand le trou entre deux grumeaux va se combler d'ou fusion des grumeaux. C'est pourquoi on va passer au double hachage. Sur cet exemple, le resultat est mediocre. Il y a deux trous mais ils sont contigus, alors qu'il faudrait dans l'ideal qu'ils soient eparpilles dans le tableau.

  double hachage :

  k1 g2 c3 e4 i5 g7 b8 j9 f11 d12 a13

  a va en 13, b en 8, c en 3, d la place 8 est prise donc va en 8+4=12, e en 4, f place 13 prise donc va en 13+12 mod 13 = 12 mais c'est pris aussi donc va en 12+12 mod 13 = 11, g va en 2, h place 2 prise, 2+9=11 pris aussi donc va finalement en 11+9 mod 13 = 7 , i va en 5, j va en 9, k va en 1.

  Notez qu'il faut que h2(x) soit premier avec la taille du tableau pour que les tentatives successsives parcourent tout le tableau. Technique 1 : t premier, h2 dans [1,t-1]. Technique 2 : t = 2^k, h2 = 2r+1, r $\in [0, 2^{k-1}[$

2 Sélections

- rappel cours: trouver le max se fait en $n-1$ comp. C'est ce qui se passe pour l'algo standard:
  max ← T[0]
  pour i de 1 a n-1
  si T[i] < max alors max ← T[i]
  Et pour un rolland-garros ? ie pour
  tant qu'il y a au moins deux candidats restants
  faire des paires et faire jouer entre les candidats de chaque paire
  les perdants sont elimines
  si le nombre de candidats etait impair,

au depart et 1 a l'arrivee, puis remarquer que chaque match elimine un candidat, et qu'il faut donc $n-1$ match pour passer de n candidats en lice a un seul.

C'est l'argument pour dire que c'est optimal (toute recherche de max a base de comparaison necessite au moins $n-1$ comparaisons): il y a n candidats max et chaque comparaison ne peut en eliminer qu'un, il en faut donc $n-1$ au moins pour n'avoir plus qu'un seul candidat.

1. (facultatif) Écrire un algorithme qui rend les éléments minimal ET maximal d'un tableau de n entiers. Faire moins de environ 2n comparaisons. Donner un code. Est-il possible de se passer d'un tableau annexe ? De ne pas modifier le tableau en entrée ?

   - facile evidemment en $2n-2$ comparaisons, puis $2n-2$ si n>1 (le min n'est le max) mais ce n'est pas optimal.

     pour n=1, 2, 3 on ne peut pas faire mieux. Mais n=4 ? On y arrive en 4 au lieu de 5, deux double comparaison, puis comparaison des gagnants et celle des perdants.

     On generalise : Faire des paires, comparer entre eux les elements de chaque paire ($n/2$ comp), mettre d'un cote les grands elements et d'un autre les petits. Comparer les grands entre eux pour avoir le max (n/2 comp), et les petits entre eux (n/2 comp) pour avoir le min. Total $(3/2)n + O(1)$ comp. Plutot que de faire d'abord toutes les comparaisons dans les paires PUIS comparer les grands et les petits entre eux, on peut faire le tout a la volee, pour chaque paire, on compare les elements de la paire entre eux, puis on compare le grand avec le plus grand des grands trouves jusqu'a present et itou pour les petits. Cela permet de donner un code "en place":

     maxmin (T,n,inout max, inout min)
     si $n < 2$
     alors erreur
     sinon
     si $T[0] < T[1]$
     alors min ← T[0] ; max ← T[1]
     sinon min ← T[1] ; max ← T[0]
     pour k de 1 a (n div 2) -1 // indice de fin de boucle, commentaire à mettre ici, cf ci-dessous
     si $T[2k] < T[2k+1]$
     alors {
     { si $T[2k] <$ min alors min ← T[2k] }
     { si $T[2k+1] >$ max alors max ← T[2k+1] } }
     sinon {
     { si $T[2k+1] <$ min alors min ← T[2k+1] }
     { si $T[2k] >$ max alors max ← T[2k] } }
     si n impair
     alors
     { si $T[n-1] <$ min alors min ← T[n-1] }
     { si $T[n-1] >$ max alors max ← T[n-1] }

     "pour k de 1 a (n div 2) -1"

     Pour trouver l'indice de fin de boucle, je conseille de poser l'équation : pour pouvoir faire le code de la boucle, il faut que les cases T[2k] et T[2k+1] soient encore dans les clous, ie que T[2k+1] soit encore dans les clous, ie que 2k+1 soit inférieur ou égal au dernier indice du tableau n-1, on veut donc $\underline{2k+1 \leq n-1}$ ie $\underline{k \leq n/2-1}$ (avec / division entière).

     Ici on peut faire un raisonnement direct : il y a n elements donc n/2 paires, puisqu'une est déjè traitée, il en reste n/2 à traiter.

     Je mettrai des commentaires dans mon code :

     pour k de 1 a (n div 2) -1 // 2k+1 <= n-1, soit k <= n/2 -1
     // raisonnement direct : n elements donc n/2 paires moins une déjà faite

     Remarque, ce coup-ci, c'est optimal: Au debut, il y 2n candidatures, n candidats max et n candidats min. A la fin, il n'en reste que 2, un candidat max et un candidat min.

     Au mieux, une comparaison elimine une candidature max et une candidature min: l'argument n'est pas assez fin car il ne donne pour minorant que $n + O(1)$.

     Plus finement: si la comparaison compare deux elements qui n'ont jamais ete compares alors elle elimine un max et un min (c'est ce que l'on peut esperer de mieux). Si elle compare deux elements dont l'un au moins a deja ete compare (par exemple compare x et y, et x ne peut plus etre min) alors il y a un resultat de la comparaison qui ne permet d'eliminer qu'une candidature (si x plus grand que y, on n'apprend rien de neuf sur x).

     On prefere donc les comparaisons du premier type qui vont plus vite. Mais on ne peut comparer deux elements qui n'ont jamais ete compares que $n/2$ fois, on ne peut pas faire donc pas plus de $n/2$ comparaisons qui elimineront deux candidatures d'un coup. Donc on peut comparer n candidatures 2 par 2 en $n/2$ comparaisons, et il faudra eliminer les autres une par une, chacune avec une comparaison, d'ou un total d'au moins $3/2n + O(1)$ comp.

     Remarque : que donne l'algo type diviser-pour-regner: "couper en deux paquets, cherche le min et le max de chaque puis comparer les min et les max entre eux ". Tout depend de n, si c'est un puissance de 2, on finit par avoir des petits paquets de taille 2, dont on determine min et max, puis on choisit min et max totaux par deux algos tournois en parallele. C'est un cas particulier de l'algo general, et il est en 3/2 de n. Si n est $3 * 2^k$, alors on finit sur des paquets de taille 3, il faut 3 comp pour en determiner le min et le max, puis a nouveau un algo tournoi en parallele pour le min et le max. Total $n + 2(n/3-1) \sim 5/3n$. On peut montrer que c'est le pire en montrant que dans ce diviser pour regner, des que $n \geq 2$, on fait moins de $5/3n-2$ comp (vrai pour 2 et 3 puis recurrence, qui marche d'ailleurs des que notre diviser pour regner fait

2. Que fait l'algorithme ci-dessous ? Quelle est, en nombre de comparaisons entre éléments du tableau, à O(1) près, la complexité au pire ? au mieux ? en moyenne (en supposant les éléments distincts, et les permutations équiprobables) ?

```
si T[0] < T[1] alors { max <- T[1]; sousmax <- T[0]; } sinon { max <- T[0]; sousmax <- T[1]; }
pour k de 2 a taille-1 faire si T[k] > sousmax
                              alors si T[k] > max alors { sousmax <- max ; max <- T[k] ; }
                                                  sinon { sousmax <- T[k] ; }
```

   - Au pire, 2n of course, ca arrive pour liste triee croissante.
     En moyenne, pour le terme k, on fait forcement une premiere comparaison, puis on en fait une deuxiemme ssi $T[k]$ est le max ou le sousmax de $T[1..k]$, ce qui arrive avec proba $2/k$, d'ou comp moyenne $\sum(1+2/k) = n + 2\,ln_e\,n + 0(1)$

3. Donner un algorithme qui rend le sous-maximal d'un tableau de n entiers en $n + \theta(ln(n)) + O(1)$ comparaisons au pire.

   - Faire un tournoi a la roland garros (si a une etape, le nombre de concurrents est impair, quelqu'un passe a l'etape suivante sans jouer). Ca fait $n-1$ comparaisons (il y a $n-1$ candidats a eliminer et chaque match en elimine un). On a le gagnant (le max). le sous max est l'un de ceux qui a perdu face au max (pas forcement en finale !, eventuellement lors du permier round), il y en a $ln_2 n$ environ. on cherche le max de ceux qui ont perdu face au grand gagnant. Note : C'est optimal encore, mais un peu casse pied a prouver

3 Dichotomie et variantes

- ne pas passer 3 heures sur le ou les exos ci-dessous. les guider rapidement pour trichotomique, puis tres rapidement pour les deux suivants (L3math)

1. (facultatif) La recherche trichotomique procède de façon similaire a la recherche dichotomique, mais en découpant les segments en 3. A chaque itération, on compare avec l'élément situé au tiers, puis si besoin avec l'élément situé au deux tiers. Comparez le nombre d'itérations, et le nombre de comparaisons en moyenne et au pire avec des recherches dichotomique et trichotomique.

   - env $ln_3 n$ iterations, ce qui est meilleur (of course, on coupe en 3 au lieu de 2 a chaque iteration). env $2ln_3 n$ comp au pire, ce qui est moins bon (il faut comparer $2\,ln\,n/ln3$ et $ln\,n/ln2$, ie $1/ln\sqrt{3}$ et $1/ln2$). En moyenne : pour une iteration donnee, on fait toujours une premiere comparaison, puis on fait une deuxieme comparaison ssi l'element est a gauche ou au milieu, soit deux chances sur trois. Donc on fait env 5/3 comp en moyenne par iteration, soit $5/3\,ln_3 n$ au total, ce qui est moins bon. (il faut comparer $(5/3)/ln3$ et $1/ln2$, ce qui revient a comparer $5/3ln3$ et $1/ln2$, ie $5ln2$ et $3ln3$ ie $ln2^5$ et $ln3^3$ ie 32 et 27)

2. (facultatif) La recherche dans un dictionnaire diffère de la dichotomie en deux points : On fait une estimation de l'emplacement du mot (on n'ouvre pas un dico au milieu si on cherche "zèbre") et on ne sait pas viser précisément un emplacement (l'humain ne sait pas ouvrir du premier coup un livre de 1000 pages à la page 500). Notez que la longueur de l'intervalle de recherche n'est pas une mesure pertinente de la distance au but, si vous cherchez "zèbre"et tombez sur "zébu", vous êtes proche, pourtant l'intervalle de recherche [abaca .. zébu] est grand. C'est pourquoi nous regarderons la distance au but du dernier mot considéré, donc ici la distance [zèbre, zébu]. On modélise la recherche en disant que si l'on vise le but depuis un certain mot à distance d, on se trompe de 10% et l'on va à distance 0.9*d ou 1.1*d. Estimez la complexité de cette recherche, comparez avec la dichotomie

   - on considere la distance du bord de l'intervalle de recherche le plus proche de l'element cherche. On a $d_{k+1} \sim d_k/10$. Puisque l'on part de $d_0 = n$ et que l'on finit avec $d_I = 1$, on obtient que le nombre d'iterations est $I \sim ln_{10} n$. Par rapport au dichotomique, on a le meme ordre de grandeur, mais on gagne un facteur multiplicatif.
     Remarque: pourquoi s'interesser a la distance au bord le plus proche de l'intervalle plutot qu'a la longueur totale de l'intervalle? Parce que l'on peut s'approcher significativement du mot cherche sans que l'intervalle de recherche ne decroisse de facon significative, exemple, je cherche le mot 762 entre 1 et 1000. Je tape a 800 (intervalle 1 a 800), puis a 770, puis a 765 (intervalle 1 a 765, il n'a presque pas diminue, pourtant on est plus proche qu'auparavant), puis a 763 puis a 762. La distance du bord le plus proche est par contre significative du "etre pres de ".

3. (facultatif) T[1..N] contient des réels de [0..1] tirés au hasard puis triés. Pour rechercher un réel r dans ce tableau, on procède comme en dichotomie, mais au lieu de regarder au milieu de l'intervalle courant [i,j], on fait une interpolation linéaire (on regarde en $i + (j-i) * (r - T[i])/(T[j] - T[i]) + O(1))$. De nouveau, la longueur de l'intervalle de recherche est non pertinente et on s'intéressera à la distance du but au bord le plus proche. On modélise cette recherche en disant que si le mot visé est à distance d d'un bord, on se trompe de $\sqrt{d}$ (une histoire d'écart-type). Estimez la complexité de cette recherche. Comparez.

   - a nouveau, ce qui est significatif, c'est la distance au bord le plus proche. On a $d_{k+1} \sim \sqrt{d_k}$, d'ou $d_k \sim n^{1/2^k}$. le nombre d'iterations est tel que $d_I = \theta(1)$, d'ou $n^{1/2^I} = \theta(1)$, soit $(1/2^I) ln\,n = \theta(1)$ soit $ln\,n = \theta(2^I)$ soit $I \sim \ln_2 \ln n$. par rapport a la recherche dichotomique, on ameliore donc l'ordre de grandeur.
     Remarque: on n'a pas la meme technique pour un humain et un ordi. Si mes N nombres sont des reels tires au hasard entre 0 et 1 puis tries, et que je veux le premier element au dela de 0.5, je vais vouloir regarder en $T[n/2]$. L'ordi pourra regarder en cette case precisemment (et pourra plus generalement regarder tres precisemment dans la case qui est celle de l'extrapolation lineaire) tandis que l'humain fera une estimation mais ne pourra etre trop precis: si vous voulez ouvrir un dico de 1000 pages a la page 500, vous n'arrivez lors de votre premier coup qu'a ouvrir a 450-550, plus generalement, on peut supposer que l'on tape la ou l'on veut avec une erreur d'une pourcentage fixe, ce qui est l'hypothese de l'enonce.
     Ensuite, si les nombres sont tires au hasard entre 0 et 1 puis tries, alors le plus proche de 0.5 se trouve en $T[n/2 + -ecart.type]$. C'est cet ecart type qui justifie la regle de progression avec une racine carree dans le dernier cas de figure.

4. (facultatif) Donnez le code de la recherche de la position du dernier élément inférieur strict à x dans un tableau trié T[0..N-1].

- On fait un test T[m] < x.

  Inclut-on T[m] dans les zones gauche et droite ? Non, l'invariant sera que l'on ne sait rien sur T[d..f], que c'est inferieur a gauche et superieur ou egal a droite, or apres la comparaison avec T[m], on sait.

  Quelle valeur pour m ? Il faut bien couper. Si la longueur est impaire 2p+1, il faut avoir deux morceaux de taille p. Si d=x+1 et f=x+2p+1, il faut trouver x+p+1, (d+f)/2 et (d+f+1)/2 conviennent. Si la longueur est paire 2p, il faut avoir deux morceaux de taille p et p-1. Si d=x+1 et f=x+2p, (d+f)/2 donne m=x+p, soit p-1 a gauche et p a droite, et (d+f+1)/2 donne m=x+p+1, soit p a gauche et p-1 a droite, les deux conviennent.

  On s'arrete quand la zone inconnue est de taille 0, ie f=d-1, et on rend f

  Note, si le tableau ne contient que des elements plus grand que x, on rend -1 pour un tableau T[0..N-1], ce qui est logique

  d = 0, f = N-1
  tant que d ≤ f
  m = (d+f)/2
  si T[m] < x
  alors d = m+1
  sinon f = m-1
  rendre f

4 Insertion dans une liste triée

L rééls triés sont rangés dans les premières cases d'un tableau T[0..M-1], avec M>L. On considère le code ci-dessous pour insérer un nouvel élément x. Quelle est la complexité au pire en nombre de comparaisons entre floats de cet algorithme ? Peut-elle être améliorée ? Si oui, comment, et quelle est la nouvelle complexité ? Itou avec la complexité au pire en nombre d'affectations de floats ?

```
L ++             \* il y a un element de plus *\
T[L-1] <- x        \* le nouvel element est range a droite *\
          \* puis il est deplace vers la gauche jusqu'a sa position finale : *\
P <- L-1
tant que (P > 0) et ( T[P] < T[P-1] )
faire     {  echanger T[P] et T[P-1]
              P-- }
```

- ameliorer en faisant une recherche par dichotomie pour determiner la position finale. Ca n'empeche pas de devoir tout deplacer, cout en comparaisons en $\theta(ln\, n)$ maiis ca reste $\theta(n)$ pour les deplacements

  F ← Dicho(...)
  L++
  T[L -1] ← x
  P ← L -1
  tant que (P > F)
  T[P -1] ← T[P]
  P - -

- un echange coute 3 comparaisons, donc 3n comp au total

  On peut s'apercevoir que l'on peut faire mieux, en faisant
  echanger x y : tmp ← x, x ← y ; y ← tmp
  (plutot que tmp ← y, y ← x ; x ← tmp)
  auquel cas l'algo va faire
  tmp ← T[l+1], T[l+1] ← T[l] ; T[l] ← tmp
  tmp ← T[l], T[l] ← T[l-1] ; T[l-1] ← tmp
  tmp ← T[l-1], T[l-1] ← T[l-2] ; T[l-2] ← tmp
  ...
  ou l'on peut constater que les suites
  T[l] ← tmp , tmp ← T[l]
  T[l-1] ← tmp tmp ← T[l-1]
  sont absurdes. L'optimisation revient en fait a eliminer ces sequences aburdes.

  On peut aussi voir directement que pour tout decaler d'un case, il suffit de sauver une extremite et de pousser un a un les elements.

  on peut faire environ n plutot que 3n mouvements, en sauvant la variable a repositionner, puis en decalant les valeurs jusqu'a liberer la place de l'element a inserer, puis en inserant le nouvel element :
  L++
  P ← L -1
  tant que (P > 0) et (T[P -1] > x)
  T[P -1] ← T[P]
  P - -
  T[P] ← x

  Note : on peut cumuler les deux optimisations.

5 Tri de 5 éléments (facultatif)

Montrez qu'un algorithme de tri à base de comparaisons de 5 éléments nécessite au moins 7 comparaisons au pire. Existe-t-il un algorithme qui effectue 7 comparaisons au pire ?

- Cf cours, un tri a base de comparaisons qui trie n elements fait au moins $ln_2n!$ comparaisons. Pourquoi?: trier revient a trouver la permutation qui remet les objets dans l'ordre. Un algo de tri doit donc distinguer toutes les permutations differentes, et il y en a n!. Au debut, toutes les permutations sont possibles. Une comparaison decoupe le paquet de comparaisons en deux paquets. la deuxieme comparaison (dont le choix depend du resultat de la premiere comparaison) decoupe chaque paquet en 2, il y a donc 4 paquets de permutations, deux permutations d'un meme paquet n'etant pas distinguees jusque-la. On continue, k comparaisons permettent de faire au plus 2^k paquets differents. Vu qu'il y a n! permutations a isoler, il faut que le nombre K de comparaisons soit tel que $2^K \geq n!$. Pour $n = 5$, on a $n! = 120$, or $2^6 = 64$ et $2^7 = 128$. Six comparaisons ne peuvent donc faire la difference qu'entre 64 cas ce qui est insuffisant pour separer les 120 permutations.

  Le but de la suite de l'exo est de reutiliser l'argument ci-dessus pour couper des branches dans la recherche exhaustive d'un algo en 7 comp. Bien utiliser l'argumentaire donc.

  Est-ce possible en 7 comparaisons ? L'argument ne donne pas de contre indication mais ne dit pas que c'est possible. On cherche donc un algo en 7 comp pour trier abcde.

  1ere comparaison entre a et b (les autres reviennent aux memes) et on peut supposer que a<b. on a donc l'ordre partiel W a<b, c, d, e

  2eme comparaison. On peut comparer c et d, ou bien c et a (les autres comparaisons reviennent au meme, utiliser aussi la symetrie min max) Si on compare c et d, on obtient par exemple c<d (l'autre cas est symetrique), et on tombe sur l'ordre partiel X0 a<b, c<d, e. Si on compare a et c, soit on tombre sur l'ordre partiel X1 $c < a < b$, d, e, soit sur l'ordre partiel X2 $a < (b, c)$, d, e.

  Si on a X1, on a 20 ordres possibles (choisir le placement de e puis celui de d), or on a droit a 5 comparaisons, ce qui permet de distinguer 32 cas. OK. Et si on a X2, on a 40 ordres possibles (choisir position de e puis celui de d puis le placement de a est obligatoire, placer alors b ; ou bien : en decidant qui de b ou de c est plus grand, on se ramene a 2 fois les possibilites qui precedent ; ou bien il y avait 120 premutations au depart, qui ont ete coupees en deux paquets egaux, W et son jumeaux, chacun englobant 60 permutations, puis W a ete separe en X1 et X2, si X1 englobe 20 permutations alors X2 en englobe $60 - 20 = 40$) et les 5 permutations ne suffiront pas a differencier tous les cas, donc il ne peux exister d'algo en 5 comp qui separe tous les cas de X2.

  Donc, la deuxieme comparaison est entre c et d, et on a alors l'ordre partiel X0 a<b, c<d, e.

  Remarque sur la derniere facon de decouper: on voit que vu qu'au depart on avait 120 permutations a separer, et qu'on pouvait couper de deux en deux jusqu'a limite de 128, on doit couper les paquets en paquets de taille egales ou quasi, car 128 est proche de 120.

  3eme comparaison : entre a et d, entre a et c ou entre a et e (autres cas symmetriques); On a droit a 4 comp. donc on ne pourra pas par la suite separer plus de 16 permutations qui n'ont pas encore ete separees. Les 30 cas de X0 doivent etre separes en deux paquets de 15, ou deux paquets de 14 et 16.

  Entre a et d, on tombe soit sur l'ordre c<d<a<b, e (5 permutations possibles, OK) soit sur l'ordre a<b, a<d, c<d, e (30-5 = 25 permutations, c'est fichu)

  Entre a et c, on obtient a<b, a<c<d, e ou configuration symetrique. Par symetrie, les deux cas se partagent la moitie des 30 permutations qui etaient possibles, soit 15 chacun, et ca reste jouable.

  Entre a et e, on tombe soit sur e<a<b, c<d, soit sur a<(b,e) c<d. Dans le deuxieme cas, si je choisis qui gagne entre b et e, je retombe sur le premier cas, donc il y a deux fois de permutations pour le deuxieme ordre partiel que pour le premier. Vu qu'elles se partagent 30 permutations, ca fait 10 pour la premiere et 20 pour la seconde. 20, c'est trop. Impasse.

  Donc la troisieme comparaison est entre a et c, on a l'ordre partiel a<b, a<c<d, e

  Quatrieme comparaison : 5 possibilites, une seule coupe les 15 comparaisons de facon equilibree en paquets de 8 et 7, c'est entre c et e. D'ou l'ordre partiel Z1 a<b, a<c<d, c<e (8 cas: pas de choix pour a, puis 4 choix pour b puis pas de choix pour c puis 2 choix), et Z2 a<b, a<c<d, e<c (15-8=7 choix).

  Trois dernieres comparaisons :

  SI on est sur Z1 : comparer d et e, puis b avec sup(d,e), puis b avec c ou avec inf(d,e) suivant resultat de la troisieme comparaison.

  SI on est sur Z2 : comparer a et e, si c'est e qui est plus grand, comparer b avec c puis au besoin avec d. Si c'est a qui est plus grand, comparer b avec c, puis b avec e ou d selon resultat de la comparaison precedente.

6 Parcours de tableaux

1. Une pile est gérée par un tableau[0..M-1] et un entier P hauteur de pile. Si P dépasse M, on alloue un nouveau tableau de longueur $2M$, on recopie le vieux tableau dans le nouveau et on continue avec le nouveau. Quel sont les coûts au pire et amorti de "empile" ? On souhaite que M reste $O(P)$, même après des dépilements. Quelle nouvelle action introduire ?

   - Coût au pire : $\theta(H)$, mais en amorti : $\theta(1)$ car avant d'avoir une operation qui coûte $\theta(H)$ (2H cases et H recopies), on empile $\theta(H)$ fois (H/2 fois). Y a qu'à dire qu'à chaque empile de coût 1, on paye 3 euros, 1 euro pour l'empile courant, et 2 euros sur le compte épargne. Quand on a un empile de coût H, on dispose de H euros sur le comte épargne (2 euros

Pour maintenir $M = \theta(H)$, il faur faire la manip inverse après des dépilements si le tableau est trop vide. Il ne faut surtout pas le faire quand la pile est a moitié vide, car on a des phénomenes de copies agrandissantes alternées sans arrêt avec des copies réduisantes en cas de empile dépile alternés. Il faut faire une coopie qui réduit la taille du tableau quand par exemple le tableau est au quart vide et on réduit dans ce cas la dimension du tableau d'un facteur 2, le coût amorti est alors constant.

PS : Ce sont les listes de Python. La structure explique pourquoi en Python, en peut empiler, depiler, et accéder directemnt au Xieme élément

2. Écrire le code de la procédure `decale (inout T[], i,j, k)` qui décale de k cases la zone T[i..j] du tableau T[0..N-1]. La zone "glisse", et telle Attila, efface tout et met des 0 sur son passage. Le code supposera qu'il n'y a pas de débordements de tableau ($0 \leq i \leq j \leq N-1 \wedge 0 \leq i+k \wedge j+k \leq N-1$). Exemples : Avec l'appel `decale(T[],4,6,4)`, le tableau [2,10,8,5,9,6,3,1,4,7,12,13] devient [2,10,8,5,0,0,0,0,9,6,3,13]. Avec l'appel `decale(T[],4,6,-2)`, le même tableau devient [2,10,9,6,3,0,0,1,4,7,12,13].

   - Notez qu'il y a un appel avec k positif et un autre avec k negatif.
     Notez qu'il y a un appel avec k plus grand que la largeur de la zone decalée et un autre avec k plus petit.

     Première chose a voir : il faut decaler avec un pour qui va dans le bon sens. Un pour decroissant si k est positif, un pour croissant si k est negatif, sinon ca buge si $|k| \leq j - i$ (prbleme d'ecrasement de valeurs).

     Deuxième chose à voir : les zéros. Il est plus simple et plus efficace de les mettre une fois le décalage fini.

     Bonne version :

     si $k > 0$
     alors
     ```
         pour m de j DECROISSANT a i
             T[m+k] ← T[m]
         pour m de i a i+k -1
             T[m] ← 0
     ```
     sinon
     si $k < 0$ alors
     ```
         pour m de i CROISSANT a j
             T[m+k] ← T[m]
         pour m de j+k+1 a j
             T[m] ← 0
     ```
     La version ci-dessous est plus lourde en ecriture
     et aussi en complexite car elle fait des affectations a 0 inutiles si $|k| \leq j - i$:
     si $k > 0$
     alors
     ```
         pour m de j DECROISSANT a i
             T[m+k] ← T[m]
             T[m] ← 0
         pour m croissant de j+1 a i+k -1
                 // il ne doit rien se passer si j + 1 > i + k − 1
             T[m] ← 0
     ```
     sinon si $k < 0$
     alors
     ```
         pour m de i CROISSANT a j
             T[m+k] ← T[m]
             T[m] ← 0
         pour m croissant de j+k+1 a i -1
                 // il ne doit rien se passer si j + k + 1 > i − 1
             T[m] ← 0
     ```

3. Écrire un algorithme qui prend en entrée un mot donné par un tableau $T[0..N-1]$ et rend vrai ssi il contient un facteur carré. Un carré est un mot $uu, u \neq \epsilon$, le mot $piftoctoccccpapaf$ contient six carrés, $abacbcabacabcbacbcabcbacabacbcabcba$ est sans carré.

   - Un alphabet est un ensemble fini dit de lettres. Un mot est une suite finie de lettres. Un mot u sera représenté par un tableau $T[0..N-1]$ de lettres. N est la longueur du mot. Par exemple, le mot $abcdda$ est représenté par le tableau $T[0..5]$ de valeur $[a, b, c, d, d, a]$.

     Il y a un mot de longueur 0, correspondant à la liste vide, dit "mot vide"

     Le mot u est facteur du mot v s'il existe des mots x et y éventuellement vides, tels que $v = xuy$. Exemple les mots $abcdda$, abc, bc et le mot vide sont facteurs du mot $abcdda$, mais aca ne l'est pas.

     Un carré est un mot de la forme uu, par exemple $blabla$

     Un mot v contient un carré s'il existe un mot non-vide u tel que uu soit facteur de v, il est sans-carré sinon. Par exemple, $abcdefdefghi$, $abccd$, $abab$ contiennent des carrés, tandis que $cbacabcbabcacbacabcacba$ est sans-carré.

     Ici, on verra qui est habile et précis. Si on s'y prend mal, on se retrouve avec des calculs d'index pénibles donc bugés. Un doué saura faire le code correct sous les ordres d'un malhabile, mais si qqun est malhabile, il est très très peu probable qu'il ait le niveau pour faire le code correct de sa malhabilité.

     On va regarder tous les facteurs qui pourraient être des carrés, et on va regarder si ce sont effectivement des carrés. Par exemple T[41..60] est-il un carré ?

     Pour caractériser un facteur, il y a plusieurs paramètres : la longueur L=20 sur l'exemple, la demi-longueur H=10 (H pour

Deux indices indépendants caractérisent la zone. Je veux éviter les calculs infernals d'indice, je n'ai pas envie de m'embêter avec des tests de parité (L doit tre pair) et donc je prends H et D (Un peu d'anticipation, regardez ce que seront les calculs avec d'autres choix. Faire le on choix ici demande certes un peu de vision et d'expérience). Prends-je le D juste avant ou juste au début, D=40 ou D=41 ? J'y vois personnelement plus clair avec D=40, mais vous pouvez préferer D=41. Je prends D=40 dernière case avant la zone (et utiliserai DD=41 pour les codes avec l'autre choix).

Je regarde donc si T[D+1,D+2*H] est un carré // si T[DD,DD+2*H-1] est un carré.

première source de variante de code : Choix des indices, H et D pour nous, mais d'autres choix sont possibles.

deuxième source de variante de code : On va faire "pour tous les facteurs", et donc (1) "pour tout D, pour tout H", versus (2) "pour tout H, pour tout D".

troisième source de variante de code :

(a) Des boucles qui vont toujours jusqu'au bout.

(b) Ou bien (optimisation) : des boucles qui s'arrêtent dès que possible grâce à des break (sortie de la boucle courante, si boucles imbriquées, on sort de tout) et du fait que return arrête toute la fonction (super-break)

(c) Ou bien simulations par des while de l'optimisation précédente

Variante 1 : pour tout D, pour tout H

Les blocs les plus à gauche sont de type [0...X], ce qui donne $\underline{D = -1}$. Le bloc le plus à droite est de taille 2 et est sur les cases N-2 et N-1, ce qui donne $\underline{D = N - 3}$

Le plus petit H est 1.

On ira comparer T[D+i] et T[D+H+i] pour i de 1 a H, donc dernière case considérée en D+2H. Il faut que D+2H soit dans les clous, ie soit plus petit que N-1, $\underline{D + 2H \leq N - 1}$, donc H va jusqu'à $\underline{(N - D - 1)/2}$ (On note que pour le départ D=-1, cela va jusqu'à N/2 ce qui est logique)

(1a)

```
AvecCarre ← faux
pour D de -1 a N-3 // existe-t-il H tel que T[D+1,D+2H] soit un carré ?
        pour H de 1 a (N-D-1) div 2 // T[D+1,D+2H] est-il un carre ?
            Carre ← vrai
            pour i de 1 a H
                si T[D+i] ≠ T[D+i+H]
                alors Carre ← faux
            si Carre alors AvecCarre ← vrai
retourner AvecCarre
```

(1b)

```
pour D de -1 a N-3 // existe-t-il H tel que T[D+1,D+2H] soit un carré ?
        pour H de 1 a (N-D-1) div 2 // T[D+1,D+2H] est-il un carre ?
            Carre ← vrai
            pour i de 1 a H
                si T[D+i] ≠ T[D+i+H]
                alors { Carre ← faux ; break }
            si Carre alors retourner vrai
retourner faux
```

(1c)

```
D = -1 ; AvecCarre ← faux
tant que non(AvecCarre) et D+2H ≤ N-1 // note : plus facile exprimé ainsi mais D+2H recalculé à chaque itération.
        H = 1
        tant que non(AvecCarre) et D+2H ≤ N-1 // itou
            i = 1 ; Carre = vrai
            tant que Carre et i ≤ H
                si T[D+i] ≠ T[D+i+H]
                alors Carre ← faux
            si Carre alors AvecCarre ← vrai
retourner AvecCarre
```

(1) avec DD : DD va de 0 à N-2, puis la condition est $DD + 2H - 1 \leq N - 1$ soit $H \leq (N - DD)/2$

```
pour DD de 0 a N-2
        pour H de 1 a (N-DD) div 2
                pour i de 0 a H-1
```

regardons maintenantla variante (2) Pour H, pour D

H va aller de 1 à N/2 (demi longeueur max d'un carre dans un tableau de N cases)

(2)
pour H de 1 a n div 2
pour D de -1 a N-2H-1 // pour DD de 0 a N-2H
...

Quelle complexite ? Trois boucles imbriquees, c'est du cubique.
Combien de tests T[D+i] $\neq$ T[D+i+H] ? : $n^3/24$
méthode 1 :
la boucle interne fait H tests
la boucle intermediare fait somme jusqu'a (N-D) div 2 de H, soit $(N-D)^2/8$ tests
la boucle exterieure fait somme jusqu'a n de $(n\text{-deb})^2/8$ = somme de $k^2/8$ (faire changement de variable) soit environ $n^3/24$
méthode 2 :
On fait les tests pour tous les triplets (i,H,D) avec $0 \leq i \leq H$ et $D+2H \leq N$, ce qui correspond a un volume en dimension 3 (faire un dessin...)
C'est un prisme, un cone, dont la base est le triangle $D + 2H \leq N$ de surface 1/4, puis $0 \leq i \leq H$ donne le cone avec la pointe en $i = H = 1/2$, $D = 0$, la hauteur du cone est 1/2, le volume d'un cone est hauteur * base / 3 ce qui nous donne le facteur 1/4 * 1/2 * 1/3 = 1/24

Est-il possible de faire mieux que cubique ? OUI !!!!!!!
on se donne H, par exemple 10, si T[14] <> T[24], alors T[5,24] n'est pas un carre, T[6,25] non plus, ... T[14,33] non plus, en fait, il est inutile de considerer les departs qui incluent 14,24 et on peur repartir de 15,25,
En fait, il y a un carre de demi longueur H ssi on peut trouver H paires successives de termes egaux a H d'ecart.
D'ou le code

```
pour H de 1 a n div 2
    cpt ← 0
    pour k de 1 a N-H
        si T[k] ≠ T[k+H]
        alors cpt ← 0
        sinon
            cpt ++
            si cpt = H alors retourner faux
retourner vrai
```

qui est quadratique

4. Écrire une procédure qui prend en entrée deux tableaux triés croissants T1[0..n1-1] et T2[0..n2-1] et écrit dans un troisième tableau T3 la fusion (union triée avec multiplicité) de ces deux listes.

- Préliminaire : on a envie de faire un "pour i1 de 0 a n1-1" et un "pour i2 de 0 a n2-1".
  L'un après l'autre ? "pour i1 finpour ; pour i2 finpour" ? Non, il ne s'agit pas de parcourir un tableau puis l'autre
  L'un imbriqué dans l'autre ? "pour i1 pour i2 finpour2 finpour1" ? Non plus, cela reviendrait à comparer tous les éléments de l'un à tous les éléments de l'autre.
  Ces deux idées sont grossièrement fausses et ne valent rien (ie pas de point si trouvé dans une copie)
  Il faut faire les deux "en parallèle" avec un while.

  On va "rendre" en effet de bord la taille du 3e tableau. On peut en fait la prédire n3=n1+n2. Nous allons faire semblant de ne pas l'avoir vu, ce qui permettra de faire un code adaptable à des variantes où n3 ne peut pas se prédire. De même, l'indice i3 peut se déduire de i1 et i2 (i3=i1+i2) mais nous allons ignorer cette possibilité pour faire un code adaptable.

```
i1 ← 0 ; i2 ← 0 ; i3 ← 0
tant que i1 < n1 et i2 < n2
    si T1[i1] < T2[i2]
    alors { T3[i3] ← T1[i1] ; i1++ ; i3++ }
    sinon { T3[i3] ← T2[i2] ; i2++ ; i3++ }
fin tant que
tant que i1 < n1 { T3[i3] ← T1[i1] ; i1++ ; i3++ }
tant que i2 < n2 { T3[i3] ← T2[i2] ; i2++ ; i3++ }
n3 ← i3
```

  Le code ci-dessus est le code standard, qui s'adaptera aux variantes.

  Autres codes :
  Les codes ci-dessous sont techniques et atypiques, la technique normale est celle ci-dessus, ne lire ci-dessous que si vous maitrisez le code ci-dessus et avez envie de voire des jongleries étonnantes.

```
i2 ← 0 ; i3 ← 0
pour i1 de 0 a n1-1
        tant que i2 < n2 et T2[i2] < T1[i1]
                { T3[i3] ← T2[i2] ; i2++ ; i3++ }
        fin tant que
        { T3[i3] ← T1[i1] ; i3++ }
fin pour
tant que i2 < n2 { T3[i3] ← T2[i2] ; i2++ ; i3++ }
```

Ou alors un grand pour sur i3 : (difficulté : écrire le booléen qui gère la boucle tant que)

```
i1 ← 0 ; i2 ← 0
pour i3 de 0 a n1+n2-1
si (i2 ≥ n2) ou (i1 < n1 et T1[i1] < T2[i2])
        alors { T3[i3] ← T1[i1] ; i1++ }
        sinon { T3[i3] ← T2[i2] ; i2++ }
```

5. Un tableau T[0..N-1] contient N objets $x_0, ..., x_{N-1}$ numérotés de 0 à N-1 (par exemple, N=13, et les objets sont des cartes à jouer). Le numéro d'un objet est donné par la fonction `Num` (On a donc $Num(x_j) = j$). On souhaite déplacer les objets de façon à ce que chaque objet x soit placé dans la case $T[Num(x)]$. Faire un premier code avec un tableau annexe $V[0..N-1]$, puis un second sans tableau annexe. Quelles sont les complexités de vos algos ?

   - Lineaire avec tableau annexe :

     ```
     pour i de 0 a N-1
             U[i] ← T[i]
     pour i de 0 a N-1
     TAB T[Num(U[i])] ← U[i]
     ```

     Lineaire sans tableau annexe :

     ```
     pour i de 0 a N-1
     tant que Num(T[i]) ≠ i
             echanger T[i] et T[Num(T[i])
     ```

     Pourquoi c'est lineaire et meme pourquoi ca termine ? Parce qu'a chaque echange, il y a un voire deux elements bien placés de plus. Le nombre d'échanges est en fait N moins le nombre de cycles de la permutation du tableau;

6. Proposez des algos pour déterminer si tous les éléments d'un tableau $T[1..N]$ sont différents.

   - On peut facilement faire des algos quadratiques, par exemple

     ```
     pour i de 0 a N-1
     TAB pour j de 0 a i-1
     TABB si T[i] = T[j] alors retourner faux
     retourner vrai
     ```

     Mais on peut faire mieux.

     Option 1 : faire une copie (pour preserver le tableau d'origine), la trier puis regarder si les elements consecutifs sont distincts. $n \ln n$ au pire en choisissant un bon tri.

     Option 2 : ranger les elements dans une table de hachage, le contexte s'y prete parfaitement puisque l'on connait le nombre d'elements. Lineaire en moyenne.

7 Quicksort (tri rapide)

- Code de Quicksort d'apres le cours :

```
Quicksort ( T[0..N-1] )
QS (T,0,n-1)

QS (T,d,f)
if f>d // else d=f, 1 element, or f=d-1, 0 element
        separate (T,d,f,pp)
        QS(T,d,pp-1)
        QS(T,pp+1,f)

Separate (T : inout, // Separates T[low..high], with pivot = T[low]
        low, high : in,
        PivotPos : out )
pivot ← T[low]
up ← low ; down ← high
```

```
        if UpGoing
        then if T[up] ≤ pivot // (up-low) + (high-down) = k
            then up++
            else T[down] ← T[up] // T[low .. up [ ≤ pivot
                down - - // T]down..high] ≥ pivot
                UpGoing ← false
        else if T[down] ≥ pivot // if UpGoing
            then down - - // then T[down] is "empty"
            else T[up] ← T[down] // else T[up] is "empty"
                up++
                UpGoing ← true
    PivotPos ← up // or PP ← down since up=down here
    T[PivotPos] ← pivot
```

1. Donnez une version de la procédure `separe` (inout T[], in d, in f, out pp) qui fait une passe de gauche à droite. Temporairement, il y aura le pivot, puis des éléments plus petits, puis des plus grands, puis des éléments non regardés.

   - Cf le premisse pour une version meilleure qui procède par le bas et par le haut.

     Ici, c'est plus facile :

     ```
     procedure separe (inout T[], in d, in f, out pp)
     finpetit = d
     pour i de d+1 a f
             si T[i] < T[d]
             alors { finpetit++
                     echange T[finpetit] et T[i] }
     echange T[finpetit] et T[d]
     pp = finpetit
     ```

2. Écrire une version itérative de Quicksort. (facultatif) Quel est l'ordre de grandeur de la hauteur maximale de la pile lors de l'exécution de cette version ? Modifier votre algorithme pour réduire la hauteur maximale de pile au pire.

   - ```
     Quicksort (T : inout , n : in)
     Pile ← vide
     empiler(1,n)
     tant que la pile n'est pas vide,
             depiler(i,j)
             si i<j
             alors
                     separe(T,i,j,pivot)
                     empiler (i,pivot-1)
                     empiler (pivot+1,j)
     ```

     Note que l'on empile des segments vides et singletons ce qui est inutile, si on ne veut pas les empiler :

     ```
     Quicksort (T : inout , n : in)
     Pile ← vide
     si n> 1
     alors empiler(1,n)
     tant que la pile n'est pas vide,
             depiler(i,j)
             separe(T,i,j,pivot)
             si pivot-1 > i alors empiler (i,pivot-1)
             si j > pivot+1 alors empiler (pivot+1,j)
     ```

   - De l'ordre de n. Pour la premiere variante (on empile les segments vides et on verra plus tard), le pire cas est quand le pivot tombe sur le dernier element, d'ou la pile contient des segments de longueur respectives (n), (0,n-1), (0,0,n-2), ... et finit par etre de hauteur n. Pour la seconde variante, (on n'empile pas les segments vides), ca reste de l'ordre de grandeur de n, le pire cas etant quand le pivot tombe toujours avant avant dernier, la pile contient des segments de longueur (n), (2,n-3), (2,2,n-6), ... et finit par etre de hauteur $n/3$

     Pour faire mieux, l'idee est tout betement, en recursif, de lancer sur le petit morceaux puis sur le grand, en iteratif, d'empiler le grand morceaux puis le petit.

     On peut galerer avant de trouver la justif que ca fait une petite pile:

     invariant, la taille d'un morceau dans la pile est plus grand que la somme des tailles de tous les morceaux au dessus, d'ou l'on tire que le kieme morceau depuis le haut est de taille au moins 2^k. La pile est alors au max de hauteur environ $ln_2 n$ au pire au lieu de 0(n)

     autre invariant : si la pile contient des bouts de tableau de tailles k0, k1, k2 .... alors $ki \leq n/2^i$

3. Donnez un algorithme de séléction inspiré de quicksort qui prend en entrée `T[0..N-1]`, un entier k et rend le k^{eme} élément (celui qui serait en position k si on trioit le tableau). Donnez la complexité au pire et estimez celle en moyenne.
     ```

- comme quicksort, mais on ne relance recursivement que sur un des cotes. Au pire, c'est toujours du $n^2/2$ comp. On estime a la louche la complexite en moyenne: le pivot tombe dans le tiers central une fois sur trois, on fait comme si il tombait dans le tiers central tous les 3 coups. Donc tous les 3 coups, la longieur de l'intervalle est divisee par au moins 3/2, donc 3 coups a n, puis 3 coups a (2/3)n puis 3 coups a $(2/3)^2$n, ce qui donne un majorant lineaire en $9n$.

  Un peu plus rigoureux: Au lieu de prendre le pivot comme premier element, je le tire au hasard parmi les elements du tableau (pour pallier eventuel defaut d'equiprobabilite des permutations, vu qu'une fonction separe peut faire perdre cette equiprobabilite). Il a une chance sur trois de tomber dans le tiers central, auquel cas on relance sur un intervalle de longueur au plus $2/3n$. Si l'on admet que la complexite moyenne sur tableau de taille n croit avec n, on obtient $C_n \leq n + 2/3 * C_n + 1/3C_{2/3n}$ d'ou $C_n \leq 3 * n + C_{2/3n}$ puis $C_n \leq 3 * n + 3 * (2/3n) + C_{(2/3)^2 n}$ puis $C_n \leq 3 * n + 3 * (2/3n) + 3 * ((2/3)^2 n) + 3 * ((2/3)^3 n) + ... = 9n$

  Code

  dans la sous fonction, on passe un k relatif ou global, ie si je cherchais le 114e, et que je suis entre 100 et 120, j'ai quoi pour k ? k=15, bof, calcul de re-indicage chiants. Garder k=114 ...

  Selection(T[0..N-1],k, out res)
  Select(T,0,k,N-1,out res)

  Select(T,i,k,j,out res) // $i \leq k \leq j$
  separe(T[], i, j, pp)
  si k < pp alors Select(T,i,k,pp-1,out res)
  si k = pp alors res $\leftarrow$ T[k]
  si k > pp alors Select(T,pp+1,k,j,out res)

4. (facultatif) SurpriseSelection procède comme QuickSelection, mais on complique la manière de chercher le pivot:
   On regroupe les n éléments par paquets de 5 éléments. On cherche le médian de chacun de ces paquets, puis on appelle récursivement SurpriseSelection pour chercher le médian de ces médians. Ce dernier est le pivot.
   Si l'intervalle de départ est de taille k, quelle taille au max et au min ont les deux intervalles résultats de la séparation ? En déduire une formule de récurrence pour la complexité au pire de cette sélection. Estimer la complexité au pire de cette sélection.

   - On suppose tous les elements differents. le pivot est plus grand que la moitie des medians de 5. Chaque median de 5 est plus grand ou egal a 3 des 5 elements de son paquet (dont lui meme), donc le pivot est plus grand qu'environ $3/5 * 1/2 = 3/10$eme des elements. Itou pour "plus petit".

     Donc la separation fait des paquets de taille entre $(3/10) * k$ et $(7/10) * k$.

     En admettant que la complexite de cet algo est croissante en n, et en negligeant les details du au fait, par exemple, que n n'est pas un multiple de 5. On obtient la relation pour la complexite au pire C_n:

     $C_n = (n/5) * M_5 + C_{n*1/5} + n + C_{n*7/10}$ ou M_5 est le nombre de comparaisons pour avoir le median de 5 elements. le premier terme pour trouver les medians de 5, le deuxieme pour trouver le median des medians, le troisieme pour la separation, le quatrieme pour l'appel recursif.

     On deroule, en posant $A = M_5/5 + 1$:

$$
\begin{aligned}
C_n &= A*n + C_{n*\frac{1}{5}} + C_{n*\frac{7}{10}} \\
&= A*n + (A*n*\tfrac{1}{5} + C_{n*\frac{1}{5}*\frac{1}{5}} + C_{n*\frac{1}{5}*\frac{7}{10}}) + (A*n*\tfrac{7}{10} + C_{n*\frac{7}{10}*\frac{1}{5}} + C_{n*\frac{7}{10}*\frac{7}{10}}) \\
&= A*n(1 + (\tfrac{1}{5} + \tfrac{7}{10})) + C_{n*\frac{1}{5}*\frac{1}{5}} + C_{n*\frac{1}{5}*\frac{7}{10}} + C_{n*\frac{7}{10}*\frac{1}{5}} + C_{n*\frac{7}{10}*\frac{7}{10}} \\
&= A*n(1 + 9/10)) \\
&\quad +(A*n*\tfrac{1}{5}*\tfrac{1}{5} + C_{n*\frac{1}{5}*\frac{1}{5}*\frac{1}{5}} + C_{n*\frac{7}{10}*\frac{1}{5}*\frac{1}{5}}) + (A*n*\tfrac{1}{5}*\tfrac{7}{10} + C_{n*\frac{1}{5}*\frac{1}{5}*\frac{7}{10}} + C_{n*\frac{7}{10}*\frac{1}{5}*\frac{7}{10}}) \\
&\quad +(A*n*\tfrac{7}{10}*\tfrac{1}{5} + C_{n*\frac{1}{5}*\frac{7}{10}*\frac{1}{5}} + C_{n*\frac{7}{10}*\frac{7}{10}*\frac{1}{5}}) + (A*n*\tfrac{7}{10}*\tfrac{7}{10} + C_{n*\frac{1}{5}*\frac{7}{10}*\frac{7}{10}} + C_{n*\frac{7}{10}*\frac{7}{10}*\frac{7}{10}}) \\
&= A*n(1 + 9/10 + (\tfrac{1}{5}*\tfrac{1}{5} + \tfrac{1}{5}*\tfrac{7}{10} + \tfrac{7}{10}*\tfrac{1}{5} + \tfrac{7}{10}*\tfrac{7}{10})) \\
&\quad +C_{n*\frac{1}{5}*\frac{1}{5}*\frac{1}{5}} + C_{n*\frac{7}{10}*\frac{1}{5}*\frac{1}{5}} + C_{n*\frac{1}{5}*\frac{1}{5}*\frac{7}{10}} + C_{n*\frac{7}{10}*\frac{1}{5}*\frac{7}{10}} + C_{n*\frac{1}{5}*\frac{7}{10}*\frac{1}{5}} + C_{n*\frac{7}{10}*\frac{7}{10}*\frac{1}{5}} + C_{n*\frac{1}{5}*\frac{7}{10}*\frac{7}{10}} + C_{n*\frac{7}{10}*\frac{7}{10}*\frac{7}{10}} \\
&= A*n(1 + 9/10 + (\tfrac{1}{5} + \tfrac{7}{10})^2) + \textstyle\sum_8 C_{...} \\
&= A*n(1 + 9/10 + (\tfrac{1}{5} + \tfrac{7}{10})^2) \\
&\quad +(\tfrac{1}{5}*\tfrac{1}{5}*\tfrac{1}{5} + \tfrac{7}{10}*\tfrac{1}{5}*\tfrac{1}{5} + \tfrac{1}{5}*\tfrac{1}{5}*\tfrac{7}{10} + \tfrac{7}{10}*\tfrac{1}{5}*\tfrac{7}{10} + \tfrac{1}{5}*\tfrac{7}{10}*\tfrac{1}{5} + \tfrac{7}{10}*\tfrac{7}{10}*\tfrac{1}{5} + \tfrac{1}{5}*\tfrac{7}{10}*\tfrac{7}{10} + \tfrac{7}{10}*\tfrac{7}{10}*\tfrac{7}{10})) \\
&\quad + \textstyle\sum_{16} C_{...} \\
&= A*n(1 + 9/10 + (\tfrac{1}{5} + \tfrac{7}{10})^2 + (\tfrac{1}{5} + \tfrac{7}{10})^3) + \textstyle\sum_{16} C_{...} \\
&= A*n(1 + 9/10 + (9/10)^2 + (9/10)^3 + (9/10)^4 + ...) \\
&= 10*A*n
\end{aligned}
$$

     (vous aurez reconnu dans $(\frac{1}{5}*\frac{1}{5} + \frac{1}{5}*\frac{7}{10} + \frac{7}{10}*\frac{1}{5} + \frac{7}{10}*\frac{7}{10})$, puis dans
     $(\frac{1}{5}*\frac{1}{5}*\frac{1}{5} + \frac{7}{10}*\frac{1}{5}*\frac{1}{5} + \frac{1}{5}*\frac{1}{5}*\frac{7}{10} + \frac{7}{10}*\frac{1}{5}*\frac{7}{10} + \frac{1}{5}*\frac{7}{10}*\frac{1}{5} + \frac{7}{10}*\frac{7}{10}*\frac{1}{5} + \frac{1}{5}*\frac{7}{10}*\frac{7}{10} + \frac{7}{10}*\frac{7}{10}*\frac{7}{10})$
     les developpements de $(\frac{1}{5} + \frac{7}{10})^2$, $(\frac{1}{5} + \frac{7}{10})^3$, etc.)

     L'algo est lineaire au pire.

8 Heapsort (tri par tas)

- Cours : on donne les algos sur les arbres, puis on les implemente en tableaux. Si le tableau commence a $T[0]$, on a fils gauche de i en $2i + 1$, fils droit en $2i + 2$ et pere en $(i - 1)\, div\, 2$. (si ça commence en $T[1]$ c'est $2i$; $2i + 1$ et $i/2$.)

  Tas de H éléments, T[0..H-1]

```
oncontinue ← vrai
tant que oncontinue et 2pos+2 ≤ H-1 {
        si T[2pos+1] < T[2pos+2]
        alors filspetit ← 2pos+1
        sinon filspetit ← 2pos+2
        si T[pos] >T[filspetit]
        alors { echanger T[pos] et T[filspetit]
                pos ← filspetit }
        sinon oncontinue ← faux }
si 2pos+1 = H-1 et T[pos] > T[2pos+1]
alors echanger T[pos] et T[2pos+1]

procedure montee (inout T,i,H)
pos ← i
tant que pos > 0 et T[(pos-1) / 2] > T[pos]
        { echanger T[pos] et T[(pos-1) /2]
        pos ← (pos-1)/2 }

procedure TriParTas (inout T,H)
pour i de 1 a H-1
montee(T,i,H)
pour i de H-1 a 1 decroissant
        echanger T[i] et T[0]
        descendre(T,0,i-1)
```

1. Complétez le tableau ci-dessous pour en faire un tas contenant les entiers de 1 a 12. Utiliser les algos du cours pour insérer une nouvelle occurrence de 3 puis retirer l'élément minimal. Faire des dessins pour justifier vos réponses.

			10	3		5					7

   - L'arbre est
     inconnu à la racine
     inconnu comme FG
     inconnu comme FD
     10 comme FG de FG
     3 comme FD de FG
     inconnu comme FG de FD
     5 comme FD de FD
     inconnu comme FG/FD de FG/FD de FG (4 cases)
     7 comme FG de FG de FD.
     On doit placer 1 à la racine
     seule possibilité pour FG : 2 car doit être plus petit que 3
     seule possibilité pour FD : 4 car doit être plus petit que 5
     seule possibilité pour FG de FD : 6 car doit être plus petit que 7
     seules possibilités pour fils de 10 : 11 et 12 (11 puis 12 ou l'inverse)
     et pour finir, fils de 3 : 8 et 9 (8 puis 9 ou l'inverse)
     et donc 1 2 4 10 3 6 5 11 12 8 9 7
     ajout de 3 : on l'insère dans la prochaine place libre, i.e. la dernière case, puis on le monte, 3 s'échange avec 6 puis avec 4 puis stop car 3 est plus grand que 1
     et donc 1 2 4 10 3 6 5 11 12 8 9 7 3
     devient 1 2 4 10 3 3 5 11 12 8 9 7 6
     puis 1 2 3 10 3 4 5 11 12 8 9 7 6
     sortie du min : 1 sort et 6 prend sa place
     6 2 3 10 3 4 5 11 12 8 9 7
     puis 6 descend : échange avec 2 puis avec 3 puis car 6 plus petit que 8 et 9
     2 6 3 10 3 4 5 11 12 8 9 7
     2 3 3 10 6 4 5 11 12 8 9 7

2. Écrire la procédure EstTas(T[0..H-1]) qui rend vrai ssi T[0..H-1] est un tas.

   - On vérifie pour i croissant que tout pere est plus petit que ses fils
     la boucle verifie pour les noeuds ayant deux fils, le cas un seul fils est gere après.
     la boucle dure tant que le fils droit est dans les clous, ie 2i+2 ≤ H-1, ie i ≤ (H-3)/2

```
VerifieTas (T,H)
pour i de 0 a (H-3)/2
        si T[i] > T[2i+1] ou T[i] > T[2i+2]
        alors rendre faux
```

rendre vrai

Variante plus simple : les fils vérifient le père.
pour i de 1 a H-1
si T[(i-1) div 2] > T[i]
alors rendre faux
rendre vrai

On peut parcourir tout les noeuds récursivement depuis la racine. Cette solution est lourde pour l'exo ici, mais est interessante car elle montre comment faire un parcours en profondeur d'un tas
verifTas (T,i,H)
verifie si OK pour T[i] et descendants
si 2i+1 > H-1 // pas de fils gauche
alors rendre vrai
sinon
si 2i+1 = H-1 //fils gauche mais pas de fils droit
alors rendre T[i] ≤ T[2i+1]
sinon
rendre T[i] ≤ T[2i+1]
et T[i] ≤ T[2i+2]
et verifTas (T,2i+1,H)
et verifTas (T,2i+2,H)

VerifieTas (T,H)
rendre verifTas(T,0,H)

Itou mais a nouveau le noeud est en charge de vérifier son père.
verifTas (T,i,H)
verifie si OK pour T[i] et descendants
si i > H-1 // pas dans l'arbre
alors rendre vrai
sinon
rendre T[i] ≥ T[(i-1)/2]
et verifTas (T,2i+1,H)
et verifTas (T,2i+2,H)

VerifieTas (T,H)
rendre verifTas(T,1,H) et verifTas(T,2,H)

3. Lorsque l'on remonte ou descend un élément dans un tas, on effectue k échanges, où k est la distance dans l'arbre entre les positions de départ et d'arrivée. Chaque échange se fait en 3 affectations, on effectue donc $3k$ affectations. Améliorez.

   - faire comme dans l'exo sur insertion d'un element dans un tableau trie. Ce qui donne par exemple pour montee:

   procedure montee (inout T,i,H)
   pos ← i
   save ← T[i]
   tant que pos > 0 et T[(pos-1) / 2] > save
   T[pos] ← T[(pos-1)/2]
   pos ← (pos-1)/2
   T[pos] ← save

4. Lorsque l'on descend un élément dans un tas depuis la racine, en règle générale, cet élément va arriver dans un niveau à distance $\theta(1)$ du fond. Utiliser cette propriété pour améliorer la complexité en nombre de comparaisons.

   - on descend l'element jusqu'en bas quoiqu'il arrive (tant que pas en bas, l'echanger avec le plus petit de ses deux fils), puis on effectue "'remonter"'. Si le tas est hauteur H et l'element doit arriver a distance h du fond, on fait $H + h$ comp au lieu de $2(H + h)$, ce qui est rentable des que $h < H/3$, or on rapelle que la moitie des elements est a $h = 1$, le quart a $h = 2$, etc. Meme si ce ne sont pas les $n/2$ plus grands qui sont a $h = 1$, on comprend que c'est en genral largement rentable.

5. Donnez la procédure `remplace(inout T,t,i,x)`, qui suppose que `T[1..t]` est un tas et que $1 \leq i \leq t$, et a pour effet de remplacer $T[i]$ par x, puis de déplacer les éléments de sorte que `T[1..t]` soit toujours un tas.

   - on fait remplace T[i] par x, puis on effectue soit "'monter"' soit "'descendre"' suivant que x est plus petit, plus grand que le vieil elmeent. ca coute du ln n.

9 Fusion de k listes

1. Montrez qu'on peut fusionner les k listes en $\sim (n * ln(k))$ comparaisons, où n est la somme des longueurs des listes. Comment s'appelle l'algorithme obtenu si les listes initiales sont des singletons ?

   - On fusionne par une sorte de tournoi :
     tant qu'il y au moins 2 listes
     rassembler les listes par paires (quitte a laisser un singleton)
     et fusionner les listes de chaque paire entre elle

     Une boucle coute moins de n comparaisons, et il y a environ $ln_2 k$ boucles.

     Si on part de singletons, on trouve un tri avec une complexite au pire en comparaisons qui est environ $n\, ln_2 n$, c'est le tri fusion.

     Remarque : onpeut essayer de generaliser la fusion de 2 en fusion de k: on repete prendre le min des k tetes de listes (k ou moins une fois que certaines listes sont epuisees). Combien cela coute-t-il ? Pour chaque sortie d'un element, on cherche un min parmi k, donc une version naive coute $n * k$.
     Puis on peut se dire que c'est bete: on cherche un min parmi k elements, que l'on sort, puis on cherche a nouveau un min parmi k elements qui sont, a un seul pres, les memes que le coup d'avant. On devrait pouvoir economiser des la deuxieme recherche. Comment faire ? Reponse, en gerant l'ensemble des k tetes de liste dans un TAS. A chaque iteration, on doit sortir le min, puis inserer un nouvel element, ce qui va couter du $ln\, k$. On obtient donc un algorithme en $\theta(n * ln\, k)$.

2. $k = 3$, on fusionne deux listes, puis le résultat avec la troisième. Quelles listes choisissez-vous de fusionner initialement ?

   - si on fusionne l1 et l2 puis le tout avec l3, la complexite est $(l1 + l2 - 1) + ((l1 + l2) + l3 - 1)$, et l'on voit que l'on a interet a fusionner d'abord les listes les plus courtes

3. Soit S une séquence de fusions. Soit A_S l'arbre valué par des listes : les feuilles sont les listes de départ, un nœud interne marque la fusion des listes fils. Donnez la complexité de S en fonction des longueurs et des positions dans A_S des listes de départ.

   - La complexite est (la somme sur tous les noeuds internes de la longueur de la liste sur le noeud) moins 1, soit somme sur toutes les feuilles descendantes du noeud, de la longueur de la liste associee a la feuille, le tout moins 1. En regroupant par liste de chaque feuille, cela donne $\sum_{l_i} |l_i| * profondeur(l_i)) - (k - 1)$. Le $k - 1$ vient de la somme des -1 (il y a exactement $k - 1$ fusions, puisque l'on part de k listes, que l'on finit avec une seule, et que chaque fusion fait diminuer le nmbres de listes de 1).

4. Montrez qu'il existe une séquence de fusion optimale dans laquelle on commence par fusionner les deux listes les plus courtes.

   - Soit A optimal. La liste la plus courte est (ou s'il y en a plusieurs: il existe une liste parmi les plus courtes qui est ...) a profondeur maximale. Par l'absurde: sinon soit B l'arbre A dans lequel on echange une liste la plus courte l_{min} avec une liste a profondeur max l_{pmax}, on a alors $Cout(B) = Cout(A) - (|l_{pmax}| - |l_{min}|) * (difference\_des\_profondeurs) < Cout(A)$, ce qui contrdit l'optimalite de A.
     Le frere de l_{min} est aussi a profondeur max, donc il y a au moins deux listes a profondeur max. Par un raisonnement similaire au precedent, la deuxieme liste la plus courte l_{deux} est aussi a profondeur max. Dans A, l_{min} et l_{deux} ne sont pas necessairement freres, mais dans ce cas on peut echanger l_{deux} et $frere(l_{min})$, ce qui ne change rien a la complexite, donc donne un arbre optimal qui fusionne les deux listes les plus courtes.

5. Donnez un algorithme simple permettant de déterminer une séquence optimale de fusions. Précisez la structure de données avec laquelle vous mémorisez les longueurs des listes qu'il vous reste à fusionner pour une fusion quelconque, et pour le tri fusion.

   - On peut donc toujours faire fusionner les deux listes les plus courtes, et apres on est ramene au meme probleme de trouver une sequence optimale mais avec une liste de moins. On peut construire l'arbre optimal comme suit:

     Foret $\leftarrow$ les l_i mis dans des feuilles, de poids $|l_i|$
     Tant que la foret a plus de deux arbre
     sortir deux arbres $A1, A2$ de poids min $p1, p2$ de E, et remettre $Cons(A1, A2)$ avec poids $p1 + p2$

     Mais en fait, il n'est pas besoin de construire explicitement un arbre. Il sert a argumenter que l'on peut commencer par fusionner les deux plus courtes, puis l'on est ramene au meme probleme, donc on peut a nouveau fusionner les deux plus courtes, etc. D"ou l'algo:

     Tant qu'il reste au moins deux listes
     Fusionner les deux listes les plus courtes

     Pour bien faire cela, il faut pouvoir retrouver les deux listes les plus courtes. Il faut donc gerer l'ensemble des listes avec leur poids, et pouvoir efficacement sortir les deux plus courtes, puis remettre le resultat de la fusion. la liste triee est chere pour la deuxieme operation. La liste non triee est chere pour la premiere. La bonne structure qui est peu chere pour les deux, c'est le tas.

     Pour le tris fusion, c'est encore mieux, on peut utiliser une file. Invariant, les elements de la file sont tries et le dernier element est plus petit que la somme des deu premiers.

10 Bucket sort (facultatif)

Pour trier N réels aléatoires de [0,1[, on prend un tableau[0..N-1] de listes chaînées initialisées à NULL, on ajoute chaque élément à trier x en case T[⌊x*N⌋], puis on trie chaque case du tableau du tableau avec le tri par séléction simple, puis on concatène les cases du tableau. Quelle est la complexité en moyenne de cet algorithme ? Cela contredit-il un théorème du cours ?

- Le placement des x, et la concaténation finale coute un temps lineaire. Quid des tris des cases ?

  Le tri par selection simple coute n(n-1)/2. Meme si ce n'est pas comme cela qu'il fonctionne, c'est le cout que l'on aurait si chaque element etait compare a tout autre une seule fois. Faisons donc semblant que ce soit le cas. Donc deu elements vont etre comparés ssi ils finissent dans la meme case. Il y a N(N-1)/2 paires d'elements. Deux elements finissent dans la meme case avecproba 1/N, donc le nombre moyen de comparaisons est 1/N * N(N-1)/2 = (N-1)/2 ce qui est lineaire. Cet algo est donc lineaire.

  Un alo du cours dit qu'un algo qui ne fait que des compraisons a une complexite au moins environ n*ln(n). Cet algo fait mieux mais il ne fait pas que des comparaisons, il fait aussi des estimations pour placer les elements, donc le theoreme ne s'y applique pas.